ENCYCLOPEDIA OF MICROCOMPUTERS

EXECUTIVE EDITORS

Allen Kent *James G. Williams*

UNIVERSITY OF PITTSBURGH
PITTSBURGH, PENNSYLVANIA

ADMINISTRATIVE EDITOR

Rosalind Kent

PITTSBURGH, PENNSYLVANIA

VOLUME 2

AUTHORING SYSTEMS FOR INTERACTIVE VIDEO TO COMPILER DESIGN

MARCEL DEKKER, INC. • NEW YORK and BASEL

Library of Congress Cataloging in Publication Data

Encyclopedia of Microcomputers.

Includes index.
1. Microcomputers–Dictionaries. I. Kent, Allen.
II. Williams, James G. III. Kent, Rosalind.
QA76.15.E52 1987 004.16'03'21 87-15428
ISBN: 0-8247-2701-0

Marcel Dekker, Inc.
270 Madison Avenue, New York, New York 10016

Current printing (last digit):
10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

CONTENTS OF VOLUME 2

CONTRIBUTORS TO VOLUME 2

HOJJAT ADELI, Associate Professor, Department of Civil Engineering, The Ohio State University, Columbus, Ohio: *Civil Engineering, Microcomputers in*

FEDERICO BARBIC, Department of Electronics, Politecnica di Milano, Milan, Italy: *Automated Office Information Systems Design*

RICHARD S. BARR, Associate Professor, Department of Operations Research and Engineering Management, Southern Methodist University, Dallas, Texas: *Benchmarking Programming Languages*

BURROUGHS CORPORATION, Corporate Research and Public Affairs, Burroughs Place, Detroit, Michigan: *Burroughs Corporation–Burroughs History*

SHI-KUO CHANG, Professor and Chairman, Department of Computer Science, University of Pittsburgh, Pittsburgh, Pennsylvania: *Coding Gray Level Binary Image Data: Image Data Compression*

PETER CROWELL, President, The Advanced Learning Systems Organization, Boulder, Colorado: *Authoring Systems for Interactive Video*

ROBERT J. COWAN, Vice President, C. Itoh Electronics, Inc., Torrance, California: *C. Itoh Electronics, Inc.*

BRIG ELLIOTT, Vice President, True BASIC, Inc., Hanover, New Hampshire: *BASIC*

IDA M. FLYNN, Lecturer, Interdisciplinary Department of Information Science, University of Pittsburgh, Pittsburgh, Pennsylvania: *COBOL*

ROGER R. FLYNN, Associate Professor, Interdisciplinary Department of Information Science, University of Pittsburgh, Pittsburgh, Pennsylvania: *Command Languages*

EDWARD H. FRAZELLE, Research Engineer, Material Handling Research Center, Georgia Institute of Technology, Atlanta, Georgia: *Automated Material Handling*

HUGH R. HOWSON, Ph.D., C.A., Associate Professor, Faculty of Management, McGill University, Montreal, Quebec, Canada: *BIOS*

DANIEL JANAL, Geltzer & Company, Inc., New York, New York: *Commodore International Limited*

EBEN LEE KENT, Information Systems New Product Manager, SRDS, Wilmette, Illinois: *Automated Program Generators*

CHARLES S. KNOX, Knox Consulting Services, St. Paul, Minnesota: *CAD/CAM, Integrating with CIM: The Management Challenge*

SARA LAFRANCE, President, Century-Analysis, Inc., Pacheco, California: *Century-Analysis, Inc.*

LEON F. McGINNIS, Associate Professor, Department of Industrial Systems Engineering, Program Manager, Manufacturing Systems, Material Handling Research Center, Georgia Institute of Technology, Atlanta, Georgia: *Automated Material Handling*

ROBERTO MAIOCCHI, Department of Electronics, Politecnico di Milano, Milan, Italy: *Automated Office Information Systems Design*

JERROLD H. MAY, Ph.D., Associate Professor and Co-Director, AIM Laboratory, Graduate School of Business, University of Pittsburgh, Pittsburgh, Pennsylvania: *Automated Forecasting*

MARLIN H. MICKLE, Ph.D., Professor of Electrical Engineering, University of Pittsburgh, Pittsburgh, Pennsylvania: *Circuits, Microprocessor*

PATRICK R. PENLAND, Ph.D., Professor, Education and Library Science, School of Library and Information Science, University of Pittsburgh, Pittsburgh, Pennsylvania: *Client-Centered Information Processing*

BARBARA PERNICI, Associate Professor, Department of Electronics, Politecnico di Milano, Milan, Italy: *Automated Office Information Systems Design*

MARK E. RORVIG, Assistant Professor, Graduate School of Library and Information Science, The University of Texas at Austin, Austin, Texas: *The Bibliographic Control of Microcomputer Software*

LAWRENCE M. SEIFORD, Associate Professor, Department of Industrial Engineering and Operations Research, University of Massachusetts, Amherst, Massachusetts: *Benchmarking Programming Languages*

KENNETH SOCHATS, Lecturer, Interdisciplinary Department of Information Science, University of Pittsburgh, Pittsburgh, Pennsylvania: *CCITT*

BRUCE STIEHM, Associate Professor, Hispanic Department, University of Pittsburgh, Pittsburgh, Pennyslvania: *Bilingual Processor*

JOHN W. TIERNEY, Professor, Chemical and Petroleum Engineering Department, University of Pittsburgh, Pittsburgh, Pennsylvania: *Chemical Engineering, Microcomputers in*

W. DAVID THORPE, Professor, Faculty of Management, McGill University, Montreal, Quebec, Canada: *BIOS*

WILLIAM G. VOGT, Ph.D., Professor of Electrical Engineering, University of Pittsburgh, Pittsburgh, Pennsylvania: *Circuits, Microprocessor*

W. M. WAITE, Professor, Department of Electrical and Computer Engineering, University of Colorado, Boulder, Colorado: *Compiler Design*

KULPATRA WETHYAVIVORN, AIM Laboratory, Graduate School of Business, University of Pittsburgh, Pittsburgh, Pennsylvania: *Automated Forecasting*

WILLIAM G. WONG, President, Logic Fusion Inc., Yardley, Pennsylvania: *Common LISP on Microcomputers*

JIAN-KANG WU, Professor, Department of Radio Electronics, University of Science and Technology of China, Hefei, People's Republic of China: *Coding Gray Level Binary Image Data: Image Data Compression*

AUTHORING SYSTEMS FOR INTERACTIVE VIDEO

BACKGROUND

Authoring systems are used in the development of interactive video (IV) systems which, in turn, are used for a variety of educational, training, retail, and presentation purposes*. IV systems generally employ a random access videodisc read by a laser and various peripheral devices to present video still and motion sequences to accomplish the system's purpose. IV (q.v) is significantly different environment from CBT, from which it is, in some ways, derived because it is principally oriented toward presentation of sound and motion or still images for the communication of information, as well as being capable of handling a variety of user inputs.

IV systems are of varying complexity, and a system of three levels has been developed to characterize them.† All refer to systems using videodiscs that permit random access to each of 54,000 video frames on the disc. Other videodiscs that do not offer random access are seldom used in IV systems.

Level 1 and 2 interactive systems require that the instructions for driving the disc be provided by either the user (Level 1) via a keypad or by a computer program on the disc (Level 2) that is interpreted by a microprocessor built into the player. Level 2 programs are written in an assembler-like computer shorthand and are usually done by a programmer. Only one commercially available authoring system will produce a level 2 program.‡

In other words, authoring systems relate almost exclusively to level 3 interactive video.

In the early days of IV, when it was time to program a computer to make the videodisc do the things the user wanted it to do, the user designed a program, wrote it, tested it, rewrote it and eventually, the job was done. That is the process that, today, we call authoring.§ It is the process of issuing, testing, and revising the instructions that present IV to the user. The development of authoring systems has taken place to accomplish several goals:

1. Reduce the cost and time involved in conventional programming techniques.
2. Permit the use of lower-cost personnel for authoring.

---

*Authoring systems are also used in development of computer-based training (CBT).

†Nebraska Videodisc Design Group, Nebraska Education Television, Lincoln, NE, is the originator of the "Nebraska levels."

‡Pides, by Professional Training Systems, Inc.

§The dictionary definition of authoring stresses the creative aspects. In IV, the creative aspects are principally in the design, scripting, and production of the materials. Authoring is essentially a mechanical task directed at implementing instructions developed in the creative process.

3. Permit the use of nonprogramming-oriented personnel (such as designers or others) in the authoring task.
4. Permit the development of sophisticated end-user systems with less cost and complexity.
5. Allow the use of sophisticated teaching algorithms such as conditional branching and artificial intelligence (AI) by personnel who are relatively unsophisticated in the programming of such techniques.

This article addresses the essential aspects of authoring systems and their attributes, without going into the details of how they work or how a team of IV system developers might use them.

COMMON TERMS

Development is the process of creating and realizing an IV system. It includes all the creative and mechanical processes in bringing a project to a successful conclusion. Among the activities subsumed in development are graphics design, scripting or writing, production, postproduction, premastering, and authoring.

Authoring is that part of the development process that involves issuing and compiling (in one sense or another) the instructions that tell a computer how to present the elements of the system to obtain the desired result.

Authoring systems are software packages that enable the author (person authoring) to issue high-level instructions regarding the system under development. "High-level" means the ability to say such powerful things as "Play from here to there, slowly", with no further explanation.

Many authoring systems are *Menu-driven*, presenting the user with a variety of menus from which authoring options are chosen. *Command-driven* authoring systems use commands or key words to accomplish the same things. Because these commands require the user to be more specific about what is to happen, when, and in what order, command-driven systems are very much like authoring languages.

Authoring languages are programming languages designed for IV that permit the programmer to issue instructions to the system that will result in the desired presentation.

Drivers are software routines in authoring systems or languages that provide the input and output facilities needed for using videodiscs, graphics, touch screens, and the other IV facilities. Most highly developed authoring software uses drivers to maintain hardware independence, that is, permits a particular system to work with many different hardware configurations.

Programming languages use expressions that are arbitrary and codelike, where an authoring language tends to use more natural symbols or terms.

Development system is the term for the software used to actually author an IV course. It is often distinct from the *run-time system*, which is the term for the software used to present the course to the end user and for managing student information. In many cases the run-time system is priced separately from the development system. Pricing policies range from a single license fee to a fee for each workstation.

AUTHORING SYSTEM CATEGORIES

Authoring systems fall into two major categories. Most are either content oriented or task oriented.

A system that permits the user to show a video motion sequence, perform graphic simulation, and select one of several options before branching might be considered oriented toward the content of that series of operations. That is, the user is able to specify a number of tasks that lead to a branch point and consider this group of instructions as a single occurrence.

Other systems concentrate on what operation is being performed now. This may be the presentation of a multiple-choice question using the touch screen, or a statement such as "Play from here to there." Or, the requirement may be to elicit an answer to a question and analyze the response. Because such systems concentrate on a specific task, they are task oriented.

What is not yet clear about this distinction is whether it makes any difference. It is possible that a content oriented system may be more productive, because the user might be able to issue instructions about a fairly large and, possibly, logical group of activities in one or a few actions. It may be that having to use a different, fairly rigid, task-oriented "template" for any action may be less productive.

Time and more rigorous methods of evaluating authoring system productivity may be required to determine this. At this early stage in the development of authoring systems, there is a real need for rigorous evaluation. There are so many competing and conflicting strategies that it is nearly impossible for potential authoring system users to know which system is most likely to meet their needs. Over 60 systems are currently marketed for micro computer-based systems, and more appear each day.

A group of instructions in an interactive system is sometimes called a "page" or a "frame," by authoring system developers. This is a concept that comes from CBT. Many system developers use these words, but they mean something different to nearly everyone. A frame for one authoring system contains only one type of task. For another, it can comprise a large number of tasks and instructions. Another problem term is "event," which suffers from the same kind of confusion. Because these terms are so vaguely defined and because their application does not affect the understanding of authoring systems directly, we will not concern ourselves with them in this article.

TYPES OF AUTHORING SYSTEMS

Among authoring systems, there are menu-driven systems, command-driven systems, and time line processors.

Menu-Driven Systems

Menu-driven systems require that you select what you want to do from a menu and may offer menus within menus that you must go through to get something done. The advantage of this technique is that it is difficult to make inadvertent mistakes, and it is easy to author an instruction even if you are not familiar with the system. On the other hand, it may be tedious and time consuming to descend through a series of menus every time you want to do the same, essential task.

Command-Driven Systems

Command-driven systems require that you learn a set of commands that must be issued to accomplish your task. If the commands are clear and logical and if there are simple and comprehensive "help" facilities available, a command-driven system can be fast and powerful, once you have mastered the language. Language is a key word, because a command-driven system is much like an authoring language. The saving grace, however, is that a command-driven authoring system is usually structured so that you can only do certain things at appropriate times. Therefore, the advantage of a command-driven authoring system is that it can be fast and powerful. The trade-off is the need to learn the language.

In some cases, an authoring system may offer both menu- and command-driven modes, and you can switch from one to another, as it suits you or as the system requires.

Time Line Processors

Time line processors are another type of system altogether. They offer the author a time line, typically measured in tenths of a second along a horizontal axis. They also list, in the vertical axis, the various types of things you may want to do. These can include video, audio, graphics, touch targets, branching, and all the other resources of the system. You, the author, must enter in each cell or space the operations that you want to occur at that time.

Time line processors do not have the disadvantages of either menus or commands, but they do require that the author understands what is going on. Generally, time line processors allow you to see an on-screen outline that tells you what instructions to insert in the time line. The advantage is very rapid authoring and revision. There is a need, however, for truly competent authoring personnel, and the instructional designer, or someone in that capacity, must prepare the outlines that guide the author. In the final analysis, this may not be a disadvantage.

WHAT IS AUTHORING?

Authoring, in some form, applies to all IV (e.g., it means developing the instructions to the disc presser or premastering house concerning chapter and picture stops on a Level 1 disc.) Such instructions imply that planning has been done previously. Even the simplest interactive videodiscs must be planned. The designer and the developer will decide how the user will obtain access to the various parts of the disc. Decisions must be made as to where everything should be on the disc and in what sequence.

Level 2 videodiscs require that the design be translated into instructions for a programmer. These instructions tell the programmer what steps to encode on the videodisc, using the instruction set required by the videodisc player's microcomputer.*

*Two manufacturers, Pioneer and Sony, provide players with level 2 capability. Neither instruction set is compatible with the other, but programs for both machines can be placed on the same videodisc.

In Level 3 systems, the design must be translated into computer files. The user may operate the system by touching a keyboard or sensitized screen.* The computer can then carry out a number of instructions, or the system may be operated by input from a variety of sensors and respond by actuating one or more peripheral devices.

Between the extremes of simplicity and complexity are many possible variations. But each of these methods of making the videodisc—computer system provide information to the user requires the developer to define the design in terms of specific objectives and, eventually, specific instructions. This design is then translated into instructions for the device(s) that will present the information that has been recorded on the videodisc.

WHY USE AN AUTHORING SYSTEM?

In the beginning, there was only computer programming. There are still IV developers who feel that programming is the only way to author an IV system. But times have changed.

Today, a variety of software developers have seen that the task of issuing commands to the computer, disc player, and other peripherals is repetitive, tedious, and time consuming. The commands required to tell a player to "play from here to there" are about the same for any situation. It should not be necessary to reprogram these instructions again and again.

If the basic commands are simple and repetitive, there are other aspects of IV system design that are not. The more instructional designers work with IV, the more sophisticated teaching algorithms they attempt and the more subtle and flexible requirements they impose.

As a simplified example, it is now standard practice to provide a graphic button to touch when the user selects an item from several on a touch screen. And, when the button is pressed, it should change color or somehow react so that you can tell your action was accepted.

This may require that the button, usually a computer graphic, be superimposed on a video signal from the disc and its color altered as necessary. The process requires one of several programming design strategies for placing a button on the screen, covering up only as much video as necessary, and so forth.

Furthermore, this process may be repeated hundreds of times during the course of a single interactive task.

This simple, frequently encountered task could require considerable programming effort. Certainly, standardized code can be designed to be included readily in a program whenever necessary, and methods can be devised to revise its location on the screen, the branching destination to be used when activated, and all the rest. But this requires a lot of work and the services of a programmer within reach at all times.

Programmers are expensive. Programming is an iterative and cranky art, and one of the requirements of IV developers is to be able to create effective systems quickly and at reasonable cost. The need to find effective authoring systems can thus be seen as part of increased productivity in system development, which reflects in the requirement for making a profit.

*Input devices include keyboard, mouse, touch screen, voice actuation, and any sensor whose input the computer can interpret.

Consistency is another factor. Both developers and customers want a series of interactive systems to be consistent in their presentation and operation. Users need to be comfortably aware that the same action will have the same effect from system to system. This is both a design and a marketing consideration.

A good authoring system will offer consistency without sacrificing power and subtlety. No benefit seems to be without its drawback, and this is true here, as well. Take, for example, the analogy of the button that changes color when touched. A given system may provide this facility, but you may not be able to change how or when it works. Perhaps all the buttons have to be round. Of course, you will have good reasons why they should be square, but the system will not permit square buttons.

Or, the system will not allow you to perform a particular task that you feel is going to increase comprehension dramatically. It just will not provide any command, utility, or task that will allow you to do what you want. That is the price you pay.

In some cases, you can avoid paying the entire price. First of all, most high-quality systems provide a great deal of flexibility. Second, most permit you to call a compiled subroutine written specifically to do the task that you want. The authoring system calls your subroutine, lets it perform its functions, and continues where it left off. This is an effective compromise.

Authoring systems, then, are computer programs designed to permit rapid, high-productivity preparation of the computer code required in Level 3 IV systems. It would be fair to describe them as very sophisticated program generators. Authoring systems are designed to be operated by a person with perception, intelligence, and skill, but they do not generally require an instructional designer or programmer. All authoring systems, each in its own way, will permit a newcomer to learn to operate them quickly and effectively.

AUTHORING LANGUAGES

Another approach to authoring is the authoring language. Authoring languages address a particular problem with traditional programming languages. BASIC, or C, or any other programming language does not recognize the existence of things like videodisc players, touch screens, and so forth. You must issue specific instructions to these devices, but no two devices— whether they are videodisc players, or graphics cards or overlay boards— accept the same instructions in the same way. So, when telling the disc to "Go here and wait." or "Play audio 2 from here to there," conventional programming languages leave something to be desired.

Authoring languages are programming languages, with all the power and flexibility that they offer, plus more. The extra power and flexibility that most authoring languages provide handle all those strange devices that make up the hardware of an IV system. They allow you to converse with the disc player directly and tell it what you want it to do. The same is true with touch screens, light pens, and so on.

Naturally, any such hybrid will have corresponding limitations. In fact, many programmers working in IV today tend to use powerful, high-level languages like C, Pascal, APL, or BASIC. Nevertheless, languages like PILOT, SUPERPILOT, PILOT PLUS, LOGOS, and others have been developed for use in presenting both CBT (represented by the acronyms CBT, CBI, CAI, CAT, etc.) and IV.

In fact, because CBT was the revolution immediately preceding IV, some programming languages and authoring systems for creating CBT have simply been expanded to include videodisc control and graphic overlay. Therefore, an authoring language gives the benefits of a system that is aware of all its facilities while still remaining a programming language, with all the benefits that it provides.

Authoring languages in general, are like more conventional programming languages but have been developed particularly for IV systems.

IV systems are distinguished by particular activities. They involve interaction with the user and, therefore, special input–output (I/O) requirements. When you add video, you must manage video and audio, as well as graphics. You must also present questions and interpret the responses in a sophisticated way.

The first authoring languages were developed for CBT. These languages offer the same things for IV that they offered for CBT, plys the additional I/O resources needed to present video. But CBT is not IV, and the extensive text-handling facilities of CBT programs are not usually required. This is because the people who developed CBT tend to think that IV is the same as CBT, but with pictures. This is not true, of course, but the authoring languages often reflect the thinking.

(This problem is found in authoring systems, as well. There are some authoring systems developed for CBT. Their IV versions betray their heritage, not always to the advantage of IV.)

PILOT, A TYPICAL AUTHORING LANGUAGE

Pilot is a typical authoring language. It was developed at the University of Washington for CBT purposes and now includes the ability to handle videodiscs and similar peripherals. A demonstration of PILOT is available, and it includes a tiny instructional program, which follows, to show you what PILOT code looks like.

```
 T: Who Wrote
  :  "The Odyssey"?
 A:
 M: HOMER
TY: Right
TN: No, Try Again
 J: @A
```

This is not a complete example of what PILOT code looks like, but it will give you an idea. T: means Type, and the colon (:) alone means more to be typed. A: means Accept input, M: is Match the input with some anticipated response. (PILOT also provides for unanticipated response.) TY: is Type if Yes, and TN: is Type if No, to provide remediation and response. The J: command is to Jump, in this case to the Accept command.

This limited example shows that although they are simple and logical, the commands are arbitrary and not obvious. Also, any rearrangement of the commands could have unpredictable results. An authoring language like PILOT would require a programmer to make it work. In that respect, it is like any programming language.

WHAT AUTHORING SYSTEMS MUST HAVE

Nat Kannan, founder and president of IMSATT Corporation, wrote an article setting forth 10 characteristics of a good authoring system* (order revised):

1. It must be easy to use, yet allow creative flexibility.
2. It must utilize interactivity in its own design.
3. It must be good at creating both simple and complex interactive systems.
4. It must support any learning theory.
5. It must be able to call and utilize other, outside subroutines.
6. It must be hardware independent.
7. It must be improved and updated constantly.
8. It should be useful at all stages of application development.
9. It should be able to incorporate higher-level systems such as AI and expert systems.
10. It should be able to accept new hardware as it is developed.

From a practical viewpoint, an IV authoring system must provide the following minimum capabilities:

1. Control of Video. This includes Play Forward and Reverse, Still Frame, Slow Motion (with variable speed), Step Forward and Reverse, and No Video. It also includes timed still frames (a slide show) and, frequently, audio with still frames.
2. Control of Audio. Audio 1, Audio 2, Stereo, and No Audio, just as with video.
3. Graphics. The ability to show a full screen graphic, or a graphic with part of the screen transparent to show video behind it, with the highest possible resolution and the largest possible selection of colors. The system should allow you to either create graphics off-line with any program of your choice or with a built-in graphics development system and to use those files in the interactive system.
4. Text (as text). Presentation of text screens has always been the major activity of CBT. IV programs must present some text, but text screens are dull, and often design objectives can be better served by using still or motion video or video through a graphic. In some IV authoring systems that come from the CBT era, text presentation is a major part of the system. In more recent, video-oriented systems, text facilities are quite limited. How much you need text presentation capability depends on your instructional strategy and on your biases.
5. Text (as graphics). There should be some simple way for the author to put a line or two of text on the screen, usually over video, without having someone create a graphic or without creating a text screen. This should also include simple shapes and the ability to change colors within a fairly wide range.

*The Videodisc Monitor, July, 1986. IMSATT is the developer of the IMSATT 2000 Authoring System.

6. Touch targets. You need to be able to define touch targets where needed—interactively—and designate branching destinations.
7. Keyboard input. You must be able to accept input from the keyboard and determine whether it is an acceptable response. This is difficult for any software to do, and how well an authoring system does it is a critical point in evaluation. Authoring system developers will tell you how sophisticated their answer analysis is, but it is not easy to get acceptance of a wide range of synonyms for an anticipated response or to deal effectively with an unanticipated one.
8. Branching. A branch is the destination or consequence related to a given input from the student or user. Most systems provide more than adequate branching but some also provide such refinements as calculated branches, conditional branches, and weighted branches, and a few also provide branching through arrays.
9. Other utilities. Some systems allow you to set indicators or flags so that you can tell when certain things should or should not happen. Can you tell, near the end of a lesson, whether condition A was chosen at the beginning? This can be really useful. Can you generate a random number so as to choose from a variety of equivalent questions? Can you set and reset variables that you can operate on for branching purposes? These are important features for any authoring system. Whether the system you buy has these features may be a matter of cost or some other factor, but if you must do without them, you will soon learn whether the choice was a wise one.
10. Management capability. You must be able to enroll students and keep records of their progress and performance. You must also be able to see how the universe of students does on each item in the system. Also, the student must be able to leave the lesson and, later, return to the place where he or she left off. Increasingly, in large corporate, educational, or governmental environments, some form of networking must be supported to permit maintenance and gathering of complex student information.

PROFESSIONAL AUTHORING ENVIRONMENT

Authoring is obviously not a task that has professional attributes. It has been a necessary evil, something that must be done to make something wonderful happen. Perhaps this is why some people think instructional designers should do the authoring. It is their penance for having the creative fun and power of developing the design.

If you sit and watch someone who is authoring, you will see a task that is, in many ways, strictly clerical. The author takes instructions that have been prepared by the instructional designer (in a flowchart or outline of the course) and translates them into the appropriate instructions that will enable the computer to implement them. It is not data entry, but at times it feels like it.

Suppose you have a sequence that plays an introduction and shows a menu that allows the user to choose either to take a tutorial or to take a test on the information it contains. It could be outlined as follows:

1. PLAY 2568 TO 2598 WITH AUDIO 1 and 2.

2. SHOW A STILL FRAME MENU (2598) WITH TWO TOUCH TARGETS.
   a. ON CHOICE ONE, GO TO 2600 AND PLAY TO 4400.
   b. ON CHOICE TWO, GO TO 4405 AND SHOW STILL FRAME QUIZ (4405).

This outline implies a large number of instructions to the system, which might be summarized as follows:

1. Ask the (videodisc) player where it is currently located and wait for a reply.
   a. If the player is not already there, tell the player to search to frame 2568.
   b. Turn on Audio 1 and 2.
   c. Tell the player to play to 2598, and check with the player until it gets there.
   d. When the player is at 2598, tell it to enter still frame mode.
2. Create a touch screen with one touch target 10 columns wide and 3 lines deep at screen coordinates row 5 and column 0 and another touch target (same size) at row 12, column 0. Wait for a touch.
   a. If the first target is touched, check to be sure where the player is, and tell it to search to 2600 and play, using Audio 1 and 2 and checking to see where it is until it gets to 4400. Then go on to further instructions (perhaps to show the quiz at 4405).
   b. If the second target is touched, check to be sure where the player is, and tell it to search to 4405 and show a still frame. Then go on to further instructions for touch targets, and so forth.

Note that this little sequence of 1 minute, 1 second, and one frame has not required any serious answer analysis, no graphics, and no manipulation of audio. The branching is also rudimentary. Yet each of these few steps implies a very much larger number of specific instructions that the computer must store, interpret, and execute when the sequence is run.

Any reasonably powerful authoring system will allow you to create the above sequence in a couple of minutes or less, if you have the correct information at hand. That information is either in the system flowchart or in some form of outline. You do not need to be an instructional designer to enter it into an authoring system, but you must know how the authoring system works and how to enter the instructions productively. This is the task of a professional author.

You can see that authoring represents the culmination of a great deal of effort and planning. It is downstream of instructional design, scripting, and production and, like any stream, receives from above all the problems and compromises that have been made during these processes. Because decisions that effect the authoring process are made from the beginning of the project, it could not hurt to have a person who knows the ins and outs of the authoring system involved at all stages. At the same time, authoring is an activity with its own demands and frequently forces change in the activities that precede it.

HOW INSTRUCTIONAL DESIGN AFFECTS AUTHORING

The capabilities of an authoring system must first be considered when an interactive system is being designed. At this stage, of course, it is possible to say that the authoring system cannot handle a particular task and that another one is needed, or it may be that the design must be tailored to meet the limitations of a specified authoring system.

The instructional designer and the graphics designer must know what the system can and cannot do. These two individuals (or functions, if you prefer) establish the strategy for and appearance of the system and are responsible for maintaining them throughout development.

One of the first tasks of the design phase is setting objectives and determining how to meet those objectives. In IV systems, this effort requires the designer to know what the authoring system can do to accomplish presentation and evaluation requirements. The designer must know that the system will handle keyboard input in such a way and what specific information must be provided for answer analysis, or that branching must be carried out in one specific manner or another. It is not necessary for the designer to know how to code this information into the computer.

There are tools now available* that assist in system development and a data base management system that permits designers and developers to maintain a constant link between objectives and strategies described by the flowchart, along with the materials required to present them and evaluate comprehension. Such tools help the developers and may, in the long run, help the author keep track of the development of the system.

The author needs to understand what is the intent and what are the strategies being implemented to achieve that intent. He or she needs to know what the things that will be authored are going to look like and how they are going to interrelate to accomplish the objectives of the system.

PRODUCTION AND POSTPRODUCTION

When a project reaches the stage of production and postproduction, most of the development has been done. Still, there will be changes and revisions; the act of putting something before the camera is often enough to reveal weakness and motivate change. To reflect this, the flowcharts and other data will be changed, too. The postproduction phase will then probably result in some additional changes.

From the authoring point of view, the result of postproduction is a list of SMPTE† time code numbers that reflect the location of specific video on the premaster and, therefore, on the videodisc. These SMPTE numbers can easily be translated into disc frame numbers, and the authoring process can begin.

---

*SCRIPTER is a data base management system for managing all aspects of script development for IV systems (Cognitive Design Technologies, Ltd., 1501 Westview Drive, Coralville, IA. 52241).

†The Society of Motion Picture and Television Engineers uses a standard form of coding to assign each television frame with a specific address in the form: HM:MM:SS:FF.

PREREQUISITES FOR BEGINNING AUTHORING

When does authoring begin? This is a question that can only be answered by the individual IV system developer. If an authoring system has a really effective flowchart utility, authoring may begin when the design is first charted. The flowchart could, conceivably, be maintained constantly in the authoring system.

When it comes time to author an interactive system, the author must have certain clearly defined resources. We have talked enough about flowcharting for you to recognize that the author must have and understand the chart.

Frame numbers can be available as early as the time the premaster is completed and, in some cases, even earlier. These frame numbers will be entered into the authoring system as required to define sequences and still frames. No truly professional authoring system would require an author to search the disc to find frames.

The author must have a list of all the significant frames on the videodisc, preferably in the order that they are to be used (which may not be the order in which they appear on the disc). Such information might well be part of the flowchart. Even if the frame numbers are part of the chart or included in some form of outline, the author may need other frame numbers to allow lead-ins to motion sequences or alternative still frames.

There also must be a list of graphics, where they are to be used, and how. There has to be a defined procedure to allow for the creation of new graphics that may not have been anticipated.

Touch screen targets must be indicated, and their branching destinations defined.

In other words, everything that is to be authored must now be defined; it can then be authored, tested, and revised. Testing is necessary to determine whether the instructions entered result in the desired presentation. (This does not refer to validation of the system.)

Different authoring systems test materials differently. Some permit the author to test each segment as it is developed: others make you run the whole course. This is because some systems distinguish between the development system and the run-time system. The development system is what the author works with and has all the bells and whistles used in system development. The run-time system is, in many cases, a subset of the development software. In cases where the run-time system is completely separate, testing can only be done by leaving the authoring system and testing the material authored in the same way a student would use it.

A good authoring system has the ability to test any part of the material that has been authored. You test the whole system or just the part that you authored in the last 10 minutes. Once testing is done, you return to the development mode (if you ever left it) and continue authoring. At that time, if the test did not work out as well as you hoped, you can reauthor the part that did not work and, if necessary, retest.

Revisions in an interactive video system are inevitable and even desirable. No one gets everything right the first time, and the best design leaves room for improvements when faced with the reality of use in the field. In the same way, a good IV authoring system must provide easy revision. It must be possible to get at any part of the system quickly and make revisions simply. It may be fair to say that any IV will be at least 50% revised before it is finished.

One of the characteristics of a good author is the ability to make judgmental changes in the way the system is presented. These small changes can make the difference between a system that is correct and one that is really slick. In this sense, there is a true creative element to the authoring task.

PROGRAMMING AS AN ALTERNATIVE TO AUTHORING SYSTEMS

Some developers of IV do not use authoring systems. They prefer to program the computer using a standard, high-level computer programming language or an authoring language that has specially developed methods for handling videodiscs, touch screens, and other IV hardware.

Why insist on computer programming when there are sophisticated authoring systems available? One answer is that in the beginning, there was no alternative. If the resources offered by Level 2 equipment were not adequate, there was no way to provide a Level 3 solution without custom programming.

Inareas where the colume of Level 3 IV is not as great as it is in the United States, custom programming has yet to be replaced by widespread use of authoring systems.

Until 1985, there was no European (i.e., PAL standard) machine capable of providing Level 2 interactivity*. All European IV has been Level 1 or 3. On the other hand, the volume of applications in Europe has been limited—partly by the lack of Level 2 and the high cost of Level 3—and there are relatively few European authoring systems available.

A compensating factor in the United Kingdom, at least in the early years, was the availability of the BBC Micro, a low-cost ($600) computer that could be used easily to drive a disc. This low cost and ease of use has meant that many British systems are custom programmed for the BBC or other microcomputer, using BASIC. As IV spreads throughout Europe, this situation will change. There are several good authoring systems already in use by and for sale by European development firms. Some of these are considering entering the U.S. market.

Another reason for custom programming lies in the ability to break new ground. IV developers are people who want to try new things to push the frontiers forward.

Why is programming better at this than an authoring system? An authoring system must, by definition, have accepted many compromises. A good system will permit the user to implement many new ideas and will provide considerable flexibility; but if the idea is to use voice recognition as an input device, simulate an oxyacetylene welding torch, or otherwise break new ground, such systems cannot do the job. Custom programming is the solution. On the other hand, authoring system developers are constantly offering new and more powerful systems that can increasingly do what could never be done before. These include AI facilities, inference engines, expert systems shells, and the like.

*The introduction of a Level 2 player by Pioneer brought Europe its first alternative to the Philips equipment, which offered only Level 1 or 3 operation.

USING CALLS TO SUBROUTINES

No authoring system is perfect. When you have found a system that meets 90%+ of your needs, it will still have shortcomings. The way a good authoring system overcomes its limitations is by permitting the author to call upon other programs for specific purposes. For example, some systems offer "bookmark" facilities, which allow users to stop at any point in a system and, when they sign on again, return to the point they left.

A system that does not provide bookmarking could be made to serve the purpose by calling a program "BKMK.COM," when the student wants to quit. This program could be written in compiled BASIC, C, or any other language supported by the system (usually a compiled language because their programs are in binary form and do not require an interpreter to be in memory when they are run).

The bookmark program would write, for example, to the student file such information as where the user is, what time it is, how long that user has been on, and so forth, depending on need. (The identity of the user would usually be in another part of this student file.) To create such a program, you must have access to a person with some programming ability. But, it is always possible to get such assistance by paying smaller fees than would be required to do the whole system.

Subroutines can also be used to let you break new ground in a limited way. If you want to use a new peripheral—a mouse or bar code reader that is not supported by your system—it may be possible to program for it. Most systems have some ability to look at an RS-232C port, a parallel port, or a game (joystick) port and get information from it. By appropriate programming, a new peripheral that talks to such a port could be recognized. You need to determine whether you really need this new thing and whether the programming effort will result in adequate results without unacceptable trade-offs. Calls to subroutines are an essential facility in any authoring system, and for these needs, there is no substitute for programming. The intelligent use of such adjuncts can make an adequate system into a great one. Programming is a valuable adjunct to an authoring system and, in some cases, is the only way to accomplish what you want to do. But the reason why authoring systems have been developed is specifically to reduce the need for expensive, time-consuming programming.

SUMMARY AND CONCLUSIONS

In general, the state of the art for authoring systems is in fragments. First, the cost of systems varies widely. There are some low-cost systems that are quite powerful and other low-cost systems of limited capability. Some of the highest cost systems do not offer some of the features of medium-cost systems.

The dichotomy between menu-driven and command-driven systems is quite wide. The most powerful command-driven systems are complex and not always easy to use or, perhaps just not easy to learn. The good menu-driven systems are usually easy to use and often quite powerful, but some show flaws that are difficult to reconcile.

Finally, there are new approaches that are surfacing all the time. These include icon-driven systems, systems that work from ASCII text files, and concurrent systems. At this time it is hard to tell what these developments will offer in the long term.

Concurrent systems, at any rate, seem to offer new options for system developers. The term concurrent usually means a system of which some part is memory resident, that is, some part of the authoring system stays in random access memory during the development and presentation of the system. This permits the developers to place additional facilities at the disposal of system users. Such facilities could include on-line help, glossaries, or access to data base management systems or other advanced software.

As far as the varied approaches to authoring environments are concerned, the marketplace will determine what survives. Menus, icons, commands, and other strategies each have their unique advantages and problems, and it is not so much a particular strategy that counts but how it is used to make it easy to author sophisticated systems.

It is very likely that new authoring systems will offer more and more facilities to integrate the development process with the authoring process, that is, ways and means to provide the equivalent of really effective flowcharting will be made part of the authoring system. So will script management, objective and strategy maintenance, and development of reports in the form of scripts, shot lists, shooting scripts, graphics requirements, and so on. Part of this will depend on the availability of larger, faster, and more powerful computers.

The 80286 computer (IBM's PC/AT and compatibles) may become the baseline machine in a few years' time. The 80386 computer will be the top-line machine (providing open architecture and operating system compatability survive the transition to the new machine), with far greater memory capacity, speed, processing power, and storage capability. This will mean that authoring systems that include many development management facilities will become feasible and a reality.

PETER CROWELL

AUTOMATED FORECASTING

INTRODUCTION

Forecasting is an essential part of planning and decision making. It is utilized in many and diverse areas such as financial planning, budgeting, sales forecasting, operation planning and control, political polling, and so forth. Various forecasting models have been developed and applied to the increasing complexity of business problems and economic environments. Fortunately, the development of computerized systems enables forecasters to access various data bases and reduce computation time. However, applications of forecasting models call for management judgment and expertise in problem identification, model selection, analysis, and interpretation of forecasting results.

This article discusses the nature of time-series data used in forecasting and five widely used forecasting models: moving averages, exponential smoothing, regression, autoregressive, and Box-Jenkins models. Emphasis will be placed on the selection of models.

TIME SERIES

A time series is a set of raw data collected chronologically and is composed of four basic elements: a trend component, a seasonal component, a cyclical component, and a random component. Monthly sales is an example of a time series. Long-time growth or decline in sales and profit suggest a trend in the data. Many products experience seasonal fluctuations with respect to demand. Certain fluctuations occur at periodic intervals, whereas other data have cyclical fluctuations that do not recur regularly. Finally, the unexplained variation between the forecast and actual value is the erratic, or random, component in a time series.

Time-series analysis is generally used when several years of data exist and when the trend is relatively stable. However, most time-series analysis techniques are inexpensive and readily available in computer packages. It is therefore important to understand the strengths and weaknesses of each technique and the types of data that are appropriate for these models.

Moving Average Forecasts

A *moving average* is an average of the values for the last N periods. The average is updated or moved each period. As the value for the new period becomes available, it is added, and the value for N+1 period back is dropped (Table 1).

The main purpose in using moving averages is to eliminate randomness in a time series. By using moving averages, trend and seasonality in the data can be eliminated. They are most commonly used for forecasting sales for short periods: a week, a month, or a quarter. The forecast consists of an average of the sales for the last N periods, where N is chosen so that the effects of seasonal factors all average out.

The following are the formulas used to calculate different moving averages, where N is the number of terms in the moving average.

Single Moving Average

$$M_t = \frac{Y_{(t-\frac{N-1}{2})} + Y_{(t-\frac{N-3}{2})} + \_\_\_\_ + Y_{(t+\frac{N-1}{2})}}{N} \tag{1}$$

where M_t is the forecast for time period t and

$$Y_{(t-\frac{N-1}{2})} = \text{actual sales for period } t-\frac{N-1}{2} \tag{2}$$

Centered Moving Average

$$M_t^1 = \frac{M_{t-1} + M_t}{2}$$

Following is a 3 × 3 moving average (or a five-term weighted moving average):

$$M_t = 0.111\ Y_{t-2} + 0.222\ Y_{t-1} + 0.333\ Y_t + 0.222\ Y_{t+1} + 0.111\ Y_{t+2} \tag{3}$$

Following is a 3 × 5 moving average (or a seven-term weighted moving average):

$$M_t = 0.067\ Y_{t-3} + 0.133\ Y_{t-2} + 0.200\ Y_{t-1} + 0.200\ Y_t + 0.200\ Y_{t+1} + 0.133\ Y_{t+2} + 0.067\ Y_{t+3} \tag{4}$$

Following is a 3 × 9 moving average (or an 11-term weighted moving average):

$$M_t = 0.037\ Y_{t-5} + 0.074\ Y_{t-4} + 0.111\ Y_{t-3} + 0.111\ Y_{t-2} + 0.111\ Y_{t-1} + 0.111\ Y_t + 0.111\ Y_{t+1} + 0.111\ Y_{t+2} + 0.111\ Y_{t+3} + 0.074\ Y_{t+4} + 0.037\ Y_{t+5} \tag{5}$$

There are other moving averages that are commonly used, such as Spencer's 15-term weighted moving average and Henderson's 5-, 9-, 13-, and 23-term weighted moving average (see Makridakis and Wheelwright).

TABLE 1 Example of Moving Averages for Sales

Year	Month	Sales (Thousands)	Single 4-Month Moving Average	Centered Moving Average	3 × 3 Moving Average	3 × 5 Moving Average	3 × 9 Moving Average
1984	Jan	264					
	Feb	260					
	Mar	262			260.07		
	Apr	259	261.25		258.30	258.40	
	May	255	259.00	260.13	256.30	255.86	
	Jun	256	258.00	258.50	253.19	252.07	250.12
	Jul	252	255.50	256.75	248.09	247.74	245.98
	Aug	240	250.75	253.13	242.31	243.13	244.16
	Sep	232	245.00	247.88	237.10	237.19	244.72
	Oct	239	240.75	242.88	231.55	236.17	248.68
	Nov	230	235.25	238.00	233.88	240.60	252.53
	Dec	205	226.50	230.88	243.87	251.67	256.78
1985	Jan	297	242.75	234.63	266.40	262.74	261.59
	Feb	289	255.25	249.00	281.39	275.34	266.40
	Mar	291	270.50	262.88	289.38	283.44	271.21
	Apr	287	291.00	280.75	286.82	286.80	276.80
	May	285	288.00	289.50	283.94	283.73	280.16
	Jun	282	286.25	287.13	280.39	280.94	281.20
	Jul	274	282.00	284.13	277.17	278.27	278.65
	Aug	275	279.00	280.50	275.17	275.93	
	Sep	276	276.75	277.88	273.84	273.73	
	Oct	272	274.25	275.50	271.95		
	Nov	271	273.50	273.88			
	Dec	265	271.00	272.25			

TABLE 2 Example of Exponential Smoothing

		Exponential Smoothing		
Month	Sales	0.05	0.25	0.4
Jan	264			
Feb	260	264.00	264.00	264.00
Mar	262	263.80	263.00	262.40
Apr	259	263.71	262.75	262.24
May	255	263.47	261.81	260.94
Jun	256	263.05	260.11	258.57
Jul	252	262.70	259.08	257.54
Aug	240	262.16	257.31	255.32
Sep	232	261.06	252.98	249.19
Oct	239	259.60	247.74	242.32
Nov	230	258.57	245.55	240.99
Dec	205	257.14	241.66	236.59

Exponential Smoothing

Exponential smoothing is a technique for obtaining a weighted moving average so that the more recent observations are assigned heavier weights (Table 2). The logic of such a weighting pattern is that the more recent periods' data are more likely to be better predictors of the next period's value than those from earlier periods.

Simple Model

$$Y_{t+1} = \alpha Y_t + (1-\alpha)\,\overline{Y}_t \tag{6}$$

where Y_{t+1} is the predicted value for the next period, α is the weight for the present period, Y_t is the present period value, and $\overline{Y}_t$ is the present period smoothed value.

The initial level of the smoothed value ($\overline{Y}_t$) can be an average of sales for the last few periods. After the first exponentially smoothed forecast is made, the present period smoothed values are then used for $\overline{Y}_t$.

General Model

$$Y_{t+1} = \alpha Y_t + \alpha(1-\alpha)\,Y_{t-1} + \alpha\,(1-\alpha)^2\,Y_{t-\alpha} + \alpha(1-\alpha)_3\,Y_{t-3} + \alpha\,(1-\alpha)^4\,Y_{t-4} + \text{———} \tag{7}$$

It may be seen from the above equation that past values of the time series Y_t are weighted in an exponentially or geometrically decreasing manner. Specifying α in the above equations may be difficult. The problem becomes magnified if it must be done for a large number of series. There are several methods that avoid this difficulty and allow for automatic specification of the parameter α, which can vary from one period to another. These methods are called adaptive response rate exponential smoothing techniques.

One adaptive method is that of Trigg and Leach. It is based on the same equation (18) but is calculated from

$$\alpha_{t+1} = \left|\frac{E_t}{M_t}\right| \tag{8}$$

where

$$E_t = \beta e_t + (1-\beta)E_{t-1} \tag{9}$$

and

$$M_t = \beta\,|e_t| + (1-\beta)\,M_{t-1} \tag{10}$$

$$e_t = Y_t - \overline{Y}_t \tag{11}$$

β is usually set at 0.1 or 0.2.

It can be seen from the above equations that α will vary with the ratio of actual to absolute error values. Equation (9) smooths the actual error, whereas Eq. (10) smooths the absolute error.

Table 3 shows the behavior of the Trigg and Leach method on our example data.

Regression

Regression is a general statistical technique through which the relationship between a *dependent* or *criterion* variable and a set of *independent* or *predictor* variables can be analyzed. For forecasting purposes, regression is used as a descriptive tool to find the best prediction equation, evaluate its prediction accuracy, and control some variables to measure their contributions to the dependent variable. Independent variables range from past values of the value to be predicted to time (e.g., year 1990, the twelfth month).

Basic Model

Values of the dependent variable are predicted from a function of the form

$$Y = A + BX + E \tag{12}$$

where Y is the dependent variable, X is the independent variable, A is the constant term, B is the marginal change in Y given one unit change in X, and E is the error term.

TABLE 3 Adaptive Response Exponential Smoothing ($\beta = 0.2$)

Year	Month	Sales	Forecast	
1984	Jan	264,000		
	Feb	260,000	264,200	0.200
	Mar	262,000	263,200	0.200
	Apr	259,000	262,960	0.200
	May	255,000	262,168	1.000
	Jun	256,000	255,000	0.826
	Jul	252,000	255,826	0.877
	Aug	240,000	252,469	0.944
	Sep	232,000	240,697	0.962
	Oct	239,000	232,330	0.499
	Nov	230,000	235,661	0.600
	Dec	205,000	232,267	0.818
1985	Jan	297,000	209,952	0.428
	Feb	289,000	247,234	0.595
	Mar	291,000	272,079	0.652
	Apr	287,000	284,420	0.660
	May	285,000	286,124	0.639
	Jun	282,000	285,405	0.564
	Jul	274,000	283,484	0.349
	Aug	275,000	280,172	0.234
	Sep	276,000	278,964	0.162
	Oct	272,000	278,483	0.004
	Nov	271,000	278,460	0.161
	Dec	264,000	277,257	0.385
1986	Jan		272,148	0.922

Note the following assumptions:

1. The model specification is given by Eq. (12).
2. The Xs are nonstochastic, and X and E are independent.
3. The error term has zero expected value and a *constant variance* across all time periods.
   a. Errors corresponding to different observations are uncorrelated.
   b. The error variable is normally distributed.

These assumptions are very important when using regression analysis. For example, assumption 2 requires that the independent variables be perfectly controlled by the researcher. The restriction of nonrandomness in the X variable is unrealistic in the study of most business problems. The results from the regression, however, may still be applied as estimations associated with a given sample, or "conditional," on the given values of the Xs. Assumption 3 may be violated, for example, if a cross section of firms in an industry is examined. There may be a reason to believe that sales figures associated with very large firms will have greater variance than those associated with small firms. Thus, the error term is unlikely to have a constant variance (homoscedastic), and heteroscedasticity arises.

If the next period's sales depend on the last period's sales, the error terms are correlated. This error process is then said to be *serially correlated*. These problems associated with the nature of data have to be corrected for better accuracy in forecasting. Heteroscedasticity and serial correlation are discussed later in this section.

The predicted value of Y is then

$$\hat{Y} = \hat{A} + \hat{B}X \tag{13}$$

where $\hat{Y}$ is the estimated value of Y, $\hat{A}$ is the estimated value of A, and $\hat{B}$ is the estimated value of B.

The difference between the actual and the estimated value of Y for each case is called the *residual* or the *error* in prediction, expressed by

$$\hat{\varepsilon} = Y - \hat{Y}$$

The regression method finds $\hat{A}$ and $\hat{B}$ so that the sum of the squared residuals is minimized, that is, min $\Sigma\,(Y - \hat{Y})^2$.

The estimators $\hat{A}$ and $\hat{B}$ are called the *ordinary least-squares estimators* (OLS) and can be obtained from

$$\hat{B} = \frac{\Sigma(X-\bar{X})\,(Y-\bar{Y})}{\Sigma(X-\bar{X})^2} \tag{14}$$

$$\hat{A} = \bar{Y} - \hat{B}\bar{X} \tag{15}$$

where

$$\bar{X} = \sum_{i=1}^{N} X_i/N$$

$$\bar{Y} = \sum_{i=1}^{N} Y_i/N$$

and N is the total number of observations.

The variability of the dependent variable Y (the total sum of squares in Y) can be partitioned into components that are (*a*) explained or accounted for by the regression line, or (*b*) an unexplained portion (sum of the squared residuals).

$$\Sigma(Y-\bar{Y})^2 = \Sigma(\hat{Y}-\bar{Y})^2 + \Sigma(Y-\hat{Y})^2 \quad (16)$$

Given the above equation (5), the measure of prediction accuracy (R^2) is given by the ratio of the explained variation in the dependent variable Y to the total variation in Y, that is,

$$R^2 = \frac{\Sigma(Y-\bar{Y})^2 - \Sigma(Y-\hat{Y})^2}{\Sigma(Y-\bar{Y})^2} \quad (17)$$

R^2 indicates the *proportion* of variation *explained* by the regression. A forecaster may be interested in the prediction accuracy based on the *absolute* amount of explained or unexplained variation. The standard error of the estimate (SEE), or the standard deviation of actual Y values from the predicted values, is given by

$$\text{SEE} = \sqrt{\frac{\Sigma(Y-\hat{Y})^2}{N-2}} \quad (18)$$

If it is assumed that actual Y values are normally distributed about the regression line, the proportion of cases that will fall between ±1, ±2, or ±3 SEE from the predicted value can be estimated. The significance of B can be tested by estimating the standard deviation of the sampling variability of B, or the standard error of B, which is given by

$$S\ (B) = \sqrt{\frac{\Sigma(Y-\hat{Y})^2/(N-2)}{\Sigma(X-\bar{X})^2}} \quad (19)$$

With a large sample size, $\hat{B}$ will approximate a normal distribution and a confidence interval, for $\hat{B}$ can be constructed using the normal distribution. If the sample size is small, $\hat{B}$ follows the t distribution with $N - 2$ degrees of freedom.

Alternatively, the null hypothesis that $B = 0$ can be tested using the F test by calculating

$$F = \frac{\Sigma(\hat{Y}-\bar{Y})^2/1}{\Sigma(Y-\hat{Y})^2/(N-2)} \quad (20)$$

with degrees of freedom 1 and $(N - 2)$.

Multiple Regression Model

The same principle as the basic case applies to the multiple regression model where there is more than one independent or explanatory variable.

$$\hat{Y} = \hat{A} + \hat{B}_1 X_1 + \hat{B}_2 X_2 + \text{———} + \hat{B}_k X_k \quad (21)$$

where $\hat{A}$s and $\hat{B}$s are derived from differentiating the sum of squares of the residuals and equating the partial derivatives to zero.

Heteroscedasticity: If the assumption that the error term has a constant variance for all observations is violated, OLS places more weight on the observations that have large error variances than on those with small error variances. Thus, when heteroscedasticity is presented, OLS estimators are unbiased but not efficient.

The multiple regression model is rewritten in matrix form:

$$Y = X\beta + e \quad (22)$$

where Y is an $n \times 1$ matrix, X is an $n \times (K+1)$, β is a $(K+1) \times 1$, e is an $n \times 1$, and $E(ee') \neq \sigma^2$ but $E\ (ee') = \sigma^2\ \Omega$ (where σ^2 is unknown but Ω is a known symmetric positive definite matrix of order n).

The following are tests for heteroscedasticity: (*a*) the Goldfield and Quant test and (*b*) the residuals test.

If heteroscedasticity is found, Aitken's generalized least squares (GLS) should be used. GLS transforms the original data Y and X into transformed variables by weighting sums of squares in the original variables by the matrix Ω^{-1}.

Serial Correlation: Serial correlation occurs in time-series studies when the errors associated with observations in a given period carry over into future time periods. A good example is predicting the growth of stock demand in which an overestimation in one year is likely to lead to overestimation in the following years. Thus,

$$E(e_i e_j) \neq 0$$

or

$$e_t = \rho e_{t-1} + u_t \quad (23)$$

where $\rho \neq 0$.

If the errors are correlated, OLS estimators are unbiased but not efficient as in the case of heteroscedasticity, and the original data need to be transformed.

The following are tests for serial correlation: (*a*) the Durbin–Watson test, (*b*) the Durbin test, and (*c*) the von Neumann test.

Corrections for serial correlations are as follows: (*a*) the Cochran-Orcutt transformation and (*b*) the Prais–Winston transformation.

After transforming the original data, GLS can be used.

Trend Projection: Trend projection using least squares is a special case of regression analysis. The main purpose of trend projection is to fit a function to a set of time-series data in which time is the independent variable.

For example, if we define x as time and y as demand,

$$y = 500 + 50x$$

If 1990 is the tenth time period in the time series, the demand forecast for 1990 can be forecast as

$$Y_{10} = 500 + 50(10)$$
$$= 1{,}000 \text{ units}$$

The values of the constant and coefficient of x (slope of the line) are least-squares estimators. Trend extrapolation for other periods can then be obtained from the estimated equation. In addition, we can use the SEE to determine the prediction interval as

$$Y_i = Y_i \pm 2\ SEE \tag{24}$$

where Y_i is the predicted value.

The results of regression, using the time period as the independent variable, generally outperform both moving average and exponential smoothing for forecasting accuracy.

Seasonal Adjustment: Seasonal adjustment techniques are based on the basic components of time series. If it is assumed that the effect of the components is *multiplicative*, a time series for Y_t will have the form

$$Y_t = L \times S \times C \times I \tag{25}$$

where L is the long-term secular trend, S is the seasonal component, C is the long-term cyclical component, and I is the random component. The objective is to estimate S.

First, isolate $L \times C$ by removing $S \times I$ from the original series Y_t. For example, suppose that Y_t consists of monthly data and a 12-month moving average $\tilde{Y}_t$ is computed. $\tilde{Y}_t$ is assumed to be free of seasonal and irregular fluctuations and is the estimate of $L \times C$. The next step is to divide the original data by this estimate to obtain a combined estimate of $S \times I$.

$$\frac{L \times S \times C \times I}{L \times C} = S \times I = \frac{Y_t}{\tilde{Y}_t} = Z_t \tag{26}$$

I is then eliminated in order to obtain the seasonal index, S, by averaging the values of $S \times I$ for the same months. The resulting values are the estimates of seasonal indexes.

Finally, Y_t can be deseasonalized by dividing each value in the series by its corresponding seasonal index.

This is a variant of a more sophisticated ad hoc procedure, the Census II method, which was developed by the Bureau of the Census of the U.S. Department of Commerce.

If it is assumed that the effect is *additive*, the time series Y_t is equal to

$$Y_t = L + S + C + I \tag{27}$$

Averaging Y_t eliminates the seasonal and random components and the result becomes

$$\tilde{Y}_t = L + C \tag{28}$$

where $\tilde{Y}_t$ is the moving average at period t. Subtracting Eq. (28) from Eq. (27) gives

$$Z_t = S + I \tag{29}$$

As in the multiplicative model, I is eliminated in order to obtain the seasonal index, S, by averaging the values of $S + I$ for the same months.

Autoregressive Model

The assumption that X is independent of the random error e is sometimes violated. This often occurs if there is a lagged dependent variable as an explanatory variable. For example, if we define Y_t as this period's demand and Y_{t-1} as the last period's demand,

$$Y_t = \beta_1 + \beta_2 Y_{t-1} + \varepsilon_t$$

The problem arises because the explanatory variable Y_{t-1} is determined by

$$Y_{t-1} = \beta_1 + \beta_2 Y_{t-2} + \varepsilon_{t-1} \tag{30}$$

Consequently, Y_{t-1} is dependent on ε_{t-1} and is in part determined by Y_{t-2} which, in turn, depends on ε_{t-2}.

Because the explanatory variable Y_{t-1} is not independent of all the error terms, the least-squares estimator is not unbiased. The maximum likelihood estimator is recommended in this case.

The maximum likelihood estimates of parameters of the probability density function (pdf) are the values that maximize the likelihood function, defined to be the joint pdf of the observed random sample and interpreted as a function of unknown parameters given the sample values.

Because Y_t depends on Y_{t-1} and this dependence is the same in all periods, the pdf of $Y_1, \ldots, Y_t$, given the unknown parameters β_1, β_2, and σ^2, is

$$f(Y_0, Y_1, \ldots, Y_t) = f(Y_0)\ f(Y_1/Y_0)\ f(Y_2/Y_1)\ \ldots\ f(Y_t/Y_{t-1})$$

Box–Jenkins Models

In the previous sections when time series are used, the models introduced are presumed to be deterministic. However, if the time series to be fore-

casted has been generated by a stochastic or random process, stochastic models should be used. These latter models provide a description of the random nature of the stochastic process that generated the sample of observations under study.

It is assumed that the observed time series $Y_1, \ldots, Y_t$ is drawn from a set of jointly distributed random variables. The purpose of these models is to explain the movement of Y_t by relating it to its past values and to a weighted sum of current and lagged random disturbances.

The following are the most commonly known stationary-stochastic models:

A moving average process of order q or MA (q) is generated by a weighted average of random disturbances of the last q periods.

$$Y_t = \mu + \varepsilon_t - \theta_1 \varepsilon_{t-1} \theta_2 \varepsilon_{t-2} \cdots \theta_q \varepsilon_{t-q} \tag{31}$$

where $E(Y_t) = \mu$, ε_t = white noise, $E(\varepsilon_t) = 0$, $E(\varepsilon^2) = \sigma^2\varepsilon$, $E(\varepsilon_t \varepsilon_{t-k}) = 0$; $k \neq 0$.

An autoregressive process of order p or AR (p) is generated by a weighted average of past observations of the last p periods, with a random disturbance in the current period:

$$Y_t = \phi_1 Y_{t-1} + \phi_2 Y_{t-2} + \cdots + \phi_p Y_{t-p} + \delta + \varepsilon_t \tag{32}$$

A mixed autoregressive-moving-average process of order (p,q) or ARMA (p,q) is represented by both types of processes.

$$Y_t = \phi_1 Y_{t-1} + \cdots + \phi_p Y_{t-p} + \delta + \varepsilon_t - \theta_t - \cdots - \theta_q \varepsilon_{t-q} \tag{33}$$

In practice, the time series are not stationary, that is, the underlying characteristics of the stochastic processes change over time. Nonstationary series can be transformed into stationary series by differentiating the time series.

Y_t is homogenous nonstationary of order d if

$$W_t = \Delta^d Y_t \tag{34}$$

is a stationary series, where

$$\Delta Y_t = Y_t - Y_{t-1} \tag{35}$$

$$\Delta^2 Y_t = \Delta Y_t - \Delta Y_{t-1} \tag{36}$$

and so forth.

A series of W_t can then be modeled as an ARMA process. If W_t is modeled as an ARMA (p,q) process, Y_t is an integrated autoregressive-moving-average process of order (p,d,q), or ARIMA (p,d,q). The problem is to choose p, d, and q, which involves examining both the autocorrelation function and the partial autocorrelation function. Important properties for identifying AR, MA, and ARMA models can be found in Table 4.

TABLE 4 AR, MA, and ARMA Models

	AR	MA	ARMA
Autocorrelation function	Infinite (damped exponential and/or damped sine waves)	Finite	Infinite (damped) exponentials and/or damped sine waves after q-p lags
	Tails off	Cut off	Tails off
Partial autocorrelation function	Finite	Infinite (dominated by damped exponentials and/or sine waves)	Infinite (dominated by damped exponentials and/or sine waves) after first p lags
	Cut off	Tails off	Tails off

Source: Box and Jenkins, *Time Series Analysis*, 1976, Chapter 3, p. 79.

Following is a sample of microcomputer packages that include forecasting capabilities. The listing is not meant to be exhaustive. The information on the listing was provided at the end of 1985.

Package:	ABSTAT
Company:	Anderson-Bill 11479 South Pine Dr. Suite 402 Parker, CO 80134 (303) 841-9755
Forecasting Techniques Include:	Simple regression Multiple regression MA Trend projection
Capacity:	Max. number of variables = 128 Max. number of items = 32,950
Operation Style:	Commands prompt Batch

Package:	Micro-TSP
Company:	Professional and Ref. Division McGraw-Hill

1221 Ave. of the Americas
New York, NY 10020
1-800-782-3737

Forecasting Techniques Include:
Single MA
Weighted MA
Exponential smoothing
Adaptive methods
Simple regression +
Multiple regression
Durbin-Watson
Coehran–Qwilt transformation
Prais–Winston transformation
Trend projection
Seasonal adjustment
MA
AR
ARMA
ARIMA

Capacity: Max. numbers of variables × data points = 10,000 (version 4.1) and 300 × 32,000 in version 5.0 (as of March 1986)

Operation Style: Commands prompt or menus
Batch

Package: NWA Statpak

Company: Northwest Analytical, Inc.
520 NW Davis St.
Portland, OR 97209
(503) 224-7727

Forecasting Techniques Include:
MA
Weighted MA
Least-squares smoothing
Simple regression
Polynomial
Multiple regression

Capacity: CPM max. number of variables = 25
MS-DOS max. number of variables = 40

Operation Style: Menus
Commands prompt

Package: SYSTAT

Company: Systat, Inc.
1127 Asbury Ave.
Evanston, IL 60202
(312) 864-5670

Forecasting Techniques Include:
Simple regression
Multiple regression
Durbin–Watson
Trend projection
Seasonal adjustment
MA
AR
ARMA
ARIMA

Capacity: Max. number of variables 25/125
Max. number of cases and items depend on disk capacity

Operation Style: Commands prompt

Package: EASI–ARIMA
TWG–ARIMA

Company: The Winchendon Group
3907 Lakota Rd.
P. O. Box 10339
Alexandria, VA 22310
(703) 960-2587

Forecasting Techniques Include:
MA
AR
ARIMA
Box–Cox transformations
Seasonal adjustments

Capacity: For APPLE max. number of variables = 250
For IBM max. number of variables = 500
max. number of cases = unlimited

Operation Style: Menus

Other Comments: EASI–ARIMA is written for users with little background in statistics.
TWA–ARIMA is for users with background in statistics.

Package: TWG–ELF

Company: The Winchendon Group
3907 Lakota Rd.
P.O. Box 10339
Alexandria, VA
(703) 960-2587

Forecasting Techniques Include: Simple regression
Multiple regression
Stepwise regression
Durbin autocorrelations

Capacity: For APPLE max. variables = 250
For IBM max. variables = 500
number of cases = unlimited

Operation Style: Menus

BIBLIOGRAPHY

Box, George E. P., and Gwilyn M. Jenkins, *Times Series Analysis: Forecasting and Control*, revised ed., Holden–Day, 1976.

Johnston, J., *Econometric Methods*, 2nd ed., McGraw-Hill, New York, 1972.

Makridakis, Spyros, and Steven C. Wheelwright, *Interactive Forecasting*, Hewlett-Packard, 1975.

Pindyek, Robert S. and Daniel L. Rubinfeld, *Econometric Models and Economic Forecasts*, 2nd ed., McGraw-Hill, New York, 1981.

Trigg, D. W., and H. Leach, "Exponential Smoothing with an Adaptive Response Rate," *Oper. Res. Q., 18*, 53–59 (1967).

Wheelwright, Steven C., and Spyros Makridakis, *Forecasting Methods for Management*, 3rd ed., Wiley, New York, 1980.

JERROLD H. MAY
KULPATRA WETHYAVIVORN

AUTOMATED MATERIAL HANDLING

1.0 INTRODUCTION

The impact of material handling on the productivity, quality, and competitiveness of manufacturing and distribution operations cannot be overstated. In a typical factory, material handling accounts for 25% of all employees, 55% of all factory space, and 87% of total production time. It is no wonder that between 15% and 70% of the total cost of a manufactured product is attributable to material handling. Quality concerns (between 3% and 5% of all handled material becomes damaged), safety concerns (60% of all industrial accidents, costing U.S. industry over $34 billion in 1984, are associated with material handling), and inventory concerns are also addressable through improved material handling practices.

Many times, improving material handling practices requires automating material handling. In fact, the opportunity for improvements through automated material handling is so dramatic that the market for automated material handling systems should exceed $9 billion by 1990. Automated material handling systems are also identified as the highest capital expense item in the automated factory—higher than computer hardware and software, machine tools and controls, programmable controllers, robots and sensors, and automated test equipment. Perhaps our resources are finally being allocated in proportion to the potential impact of the technology on the productivity, quality, and competitiveness of our manufacturing and distribution operations.

Automated material handling systems are the subject of this article. Section 2.0 describes automated material transport, Section 3.0 describes automated storage and retrieval, Section 4.0 describes integrated storage and handling practices, and Section 5.0 describes automated material control.

This presentation is by no means a comprehensive treatment of automated material handling. Instead, it addresses most of the major equipment types for unit load handling and does not address bulk handling (grain, coal, oil, etc.). The emphasis is on technology description and identification of the role of microcomputers in automated material handling. The interested reader should consult the bibliography for more detailed descriptions.

2.0 AUTOMATED MATERIAL TRANSPORT

The requirement for material flow is universal. Without material flow, nothing happens in a factory, warehouse, distribution center, or retail store. The opportunities for automating material flow are therefore virtually limitless (although the economic justification may be a limiting factor). As the following presentation illustrates, different material transport technologies are probably best viewed as points on a continuum of technological evolution.

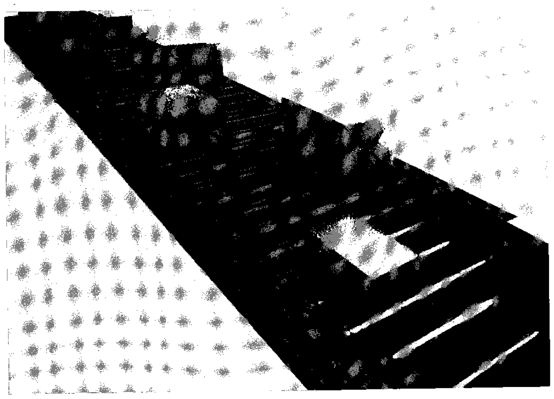


FIGURE 1 Traditional live roller conveyor.

FIGURE 2 Transporter conveyor.

2.1 Conveyors

The most common unit load transport technology is the conveyor, which may take many forms. Conveyors provide a "roadway" along which a unit load moves. The force moving the unit load may be provided by gravity, or by the conveyor itself. Above-floor, powered conveyors may use a sliding belt, slats, chains, or live rollers to apply the motive force directly to the unit load and, typically, are located at a height of 3 to 5 feet above floor level. Figure 1 illustrates a traditional live roller conveyor.

Conveyors are an excellent means for moving unit loads at a high rate from one point to another. As the number of origins and/or destinations increases, the complexity of traffic control increases. The loads must be tracked, occasionally with mechanical switches, but more often with photoelectric sensors. Microcomputers, in the form of programmable controllers, are an integral part of conveyor systems.

Transporter conveyors are a popular technology for managing the flow of work-in-process materials in a manufacturing cell. Transporter conveyors permit the automatic dispatching of loads to particular workstations located along the conveyor. As the load reaches the designated workstation, it is automatically diverted. Upon completing its required processing, the load is placed on a return conveyor. Figure 2 shows a transporter conveyor.

Sortation systems often employ high-speed conveyors with special load-carrying platforms, or tilt trays. Packages are deposited on the tilt trays at rates of 120/per minute or higher. As the tray reaches the appropriate delivery chute, the tray tilts, dropping the package into the chute. These systems employ machine-readable labels on the packages, high-speed readers, and programmable controllers. Figure 3 shows a tilt-tray sorter.

2.2 Alternative Conveyor Configurations

Although above-floor conveyors are ideal for many applications, they do have some undesirable attributes. They present physical obstructions that may limit the movement of people and vehicles, they are fairly expensive to reconfigure, and their control systems become unpleasantly complicated when there are many origins and destinations for unit loads and a random distribution of unit load routes.

Two alternative configurations that overcome the problem of physical obstruction are towlines and power-and-free conveyors. In these systems, a continuous chain moves around a closed path to provide the force required to move unit loads.

Towline conveyor systems have the chain running in a slot in the floor. Wheeled vehicles are pulled along by a pin that engages the chain. Power-and-free conveyors have the chain running in a channel suspended overhead and employ carriers or trolleys that ride on a separate rail structure. The trolleys may be diverted to an unpowered (or "free") segment of track to allow the unit load to be unloaded, loaded, or otherwise manipulated.

Both towlines and power-and-free conveyors permit the unit load to be routed to a specific location on the chain path and to be diverted and stopped at that station. The system for engaging/disengaging the chain may be completely mechanical, or electromechanical, in which case programmable controllers will be required.

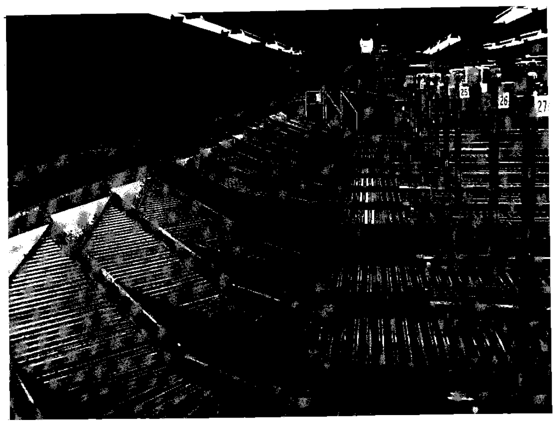


FIGURE 3 Tilt-tray sorter.

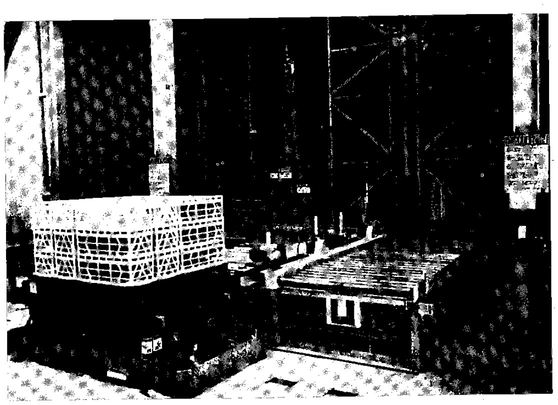


FIGURE 4 Unit load AGV in front of an AS/RS (courtesy of Eaton-Kenway Corp.).

Towlines and power-and-free conveyors do not have the same movement rate capacity as traditional above-floor conveyors, but they do eliminate the problem of physical obstructions. Note that towlines do present some additional congestion and collision problems, because they use a wheeled vehicle on the floor. Power-and-free conveyors, on the other hand, are overhead and need not follow any aisle structure.

A limitation of both towlines and power-and-free conveyors is the inflexible path of the chain. The chain must be a continuous loop, and even though the carriers or trolleys may be switched from one chain to another (i.e., from one loop to another), there is limited routing flexibility. The material transport technologies described in the next two sections overcome this limitation by having the unit load carriers provide the motive force, rather than having it provided by the path.

2.3 Automated Guided Vehicles

An automated guided vehicle, or AGV, is essentially a driverless industrial truck. Is is a steerable, wheeled vehicle, driven by electric motors using storage batteries, and it follows a predefined path along an aisle. AGVs may be designed to operate as a tractor, pulling one or more carts, or may be unit load carriers. Figure 4 illustrates a unit load AGV, which is the most common type in manufacturing and distribution.

The path followed by an AGV may be a simple loop or a complex network, and there may be many designated load/unload stations along the path. The vehicle incorporates a path-following system, typically electromagnetic, although some optical systems are in use.

With an electromagnetic path-following system, a guide wire that carries a radio frequency (RF) signal is buried in the floor. The vehicle employs two antennae, so that the guide wire can be bracketed. Changes in the strength of the received signal are used to determine the control signals for the steering motors so that the guide wire is followed accurately. When it is necessary for the vehicle to switch from one guide wire to another (e.g., at an intersection), two different frequencies can be used, with the vehicle being instructed to switch from one frequency to the other. Obviously, a significant level of both analog and digital electronic technology is incorporated into the vehicle itself. In addition, the vehicle routing and dispatching system will not only employ programmable controllers, but minicomputers or even mainframe computers.

The electromagnetic path-following system may also support communication between a host control computer and the individual vehicles. In systems with a number of vehicles, the host computer may be responsible for both the routing and dispatching of the vehicles and collision avoidance. A common method for collision avoidance is zone blocking, in which the path is partitioned into zones, and a vehicle is never allowed to enter a zone already occupied by another vehicle. This type of collision avoidance involves a high degree of active, host computer–directed control.

Optical path-following systems use an emitted light source and track the reflection from a special chemical stripe painted on the floor. In a similar fashion, codes can be painted on the floor to indicate to the vehicle that it should stop for a load/unload station. Simple implementations will require all actions of the vehicle to be preprogrammed, for example, stop and look for a load, stop and unload, or proceed to the next station. The vehicles

typically will not be under the control of a host computer and must therefore employ some method for collision avoidance other than zone blocking. Proximity sensors on the vehicles permit several vehicles to share a loop without colliding.

The state of the art in AGVs is advancing rapidly. There are now "smart" vehicles that can navigate for short distances without an electromagnetic or optical path. Similarly, vehicles are being equipped with sufficient on-board computing capability to manage some of the routing control and dispatching functions. Current developments in the field are leading to path-free, or "autonomous" vehicles, which do not require a fixed path and are capable of "intelligent" behavior. Much of the work currently under way in the AGV field is driven by the availability of small, powerful, but relatively inexpensive microcomputers.

2.4 Self-Powered Monorails

Self-powered monorails (SPMs), also referred to as automated electrified monorails (AEMs) resemble overhead power-and-free conveyors in much the same way that AGVs resemble towline conveyors. An SPM moves along an overhead monorail, but rather than being driven by a moving chain, it is driven by its own electric motor. The SPM provides many of the benefits of both overhead power-and-free conveyors and AGVs. It is overhead, so it does not create obstructions on the factory floor, and self-powered and programmable, so it has a great deal of route flexibility. In contrast to the AGV, it may use a bus bar to provide the electric current required by the drive motors, rather than carrying a storage battery. Figure 5 shows a typical SPM.

Another distinctive feature of SPMs is that they require no sophisticated path-following system, because the path must conform to the monorail structure. On the other hand, path selection, for example, at an intersection or branch point, requires coordination between a vehicle-tracking function and a physical track-switching function. With an AGV using wire guidance, the path selection may be accomplished by the vehicle itself, simply by switching frequencies.

SPMs have been designed to interface with a wide variety of production equipment. The carriers can be switched from one monorail line to another and can even be raised or lowered between two different levels. As with AGVs, the SPM employs a high degree of computer control.

2.5 Microcomputer Applications in Material Transport

Any automated material transport system will employ microcomputers, in the form of programmable controllers, as the interface between the logical control system (the digital world) and the physical devices such as sensors and actuators (the analog world). Very often, the programmable controllers will communicate with another computer, which can be described as a "cell host."

The cell host will manage the interaction between a number of related physical devices. For example, there might be a cell host coordinating all of the activities of the physical devices employed in a transporter conveyor network. The cell host is not only concerned with the physical actions but also with issues such as job routing, job priorities, and workstation availability. In the past, the cell host typically has been a minicomputer, but as the power of microcomputer systems increases, they will find wider application in the cell host function.

FIGURE 5 SPM carriers.

3.0 AUTOMATED STORAGE AND RETRIEVAL

This section describes automated storage and retrieval systems (AS/RS) for both large and small loads. Large loads are considered to be pallet loads (or unit loads) of material, and small loads are defined to be tote pan size and smaller.

3.1 Automated Storage Systems for Large Loads

An AS/RS for large loads is commonly referred to as a unit load AS/RS. It is defined by the AS/RS product section of the Material Handling Institute as a storage system that uses fixed-path storage and retrieval (S/R) machines running on one or more rails between fixed arrays of storage racks (see Figs. 6 and 7).

A unit load AS/RS usually handles loads in excess of 1,000 pounds and is used for raw material, work-in-process, and finished goods. The number of systems installed in the United States is in the hundreds, and installations are commonplace in all major industries.

A typical AS/RS operation involves the S/R machine picking up a load at the front of the system, transporting the load to an empty location, depositing the load in the empty location, and returning empty to the input/output (I/O) point. Such an operation is called a single command (SC) operation. Single commands accomplish either a storage or a retrieval between successive visits to the I/O point. A more efficient operation is a dual

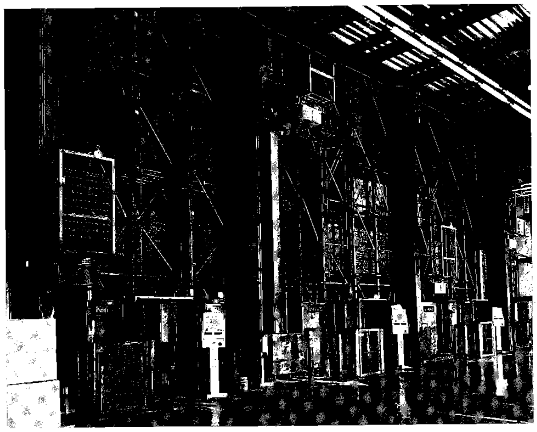


FIGURE 6 Typical unit load AS/RS (courtesy of Eaton-Kenway Corp.).

command (DC) operation. A DC involves the S/R machine picking up a load at the I/O point, traveling loaded to an empty location (typically the closest empty location to the I/O point), depositing the load, traveling empty to the location of the desired retrieval, picking up the load, traveling loaded to the I/O point, and depositing the load. The key idea is that in a DC, two operations, a storage and a retrieval, are accomplished between successive visits to the I/O point.

A unique feature of the S/R machine travel is that vertical and horizontal travel occur simultaneously. Consequently, the time to travel to any destination in the rack is the maximum of the horizontal and vertical travel times required to reach the destination from the origin. Horizontal travel speeds are on the order of 500 feet per minute; vertical, 120 feet per minute.

The typical unit load AS/RS configuration, if there is such a thing, would include unit loads stored one deep (i.e., single deep), in long narrow aisles, each of which contains a S/R machine. The one I/O point would be located at the lowest level of storage and at one end of the system.

More often than not, however, one of the parameters defining the system is atypical. The possible variations include the depth of storage, the number of S/R machines assigned to an aisle, and the number and location of I/O points. These variations are described in more detail below.

When the variety of loads stored in the system is relatively low, throughput requirements are moderate to high, and the number of loads to be stored

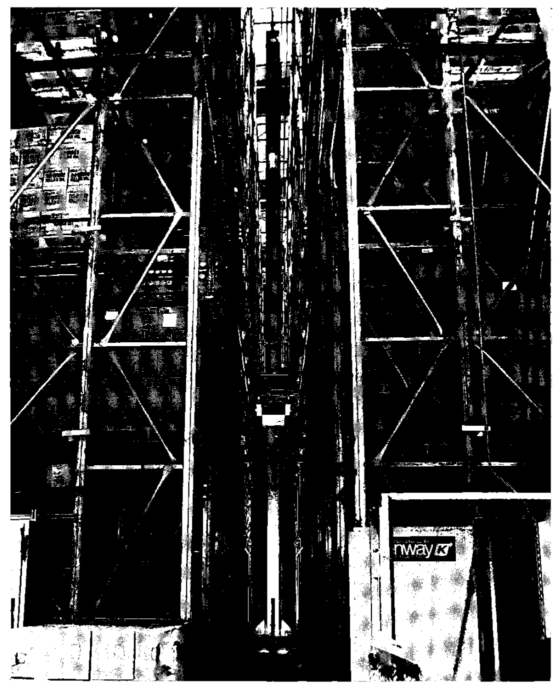


FIGURE 7 Typical unit load AS/RS (courtesy of Eaton-Kenway Corp.).

is high, it is often beneficial to store loads more than one deep in the rack. Alternative configurations include

- Double deep storage with single-load width aisles. Loads of the same stock-keeping unit (SKU) are typically stored in the same location. A modified S/R machine is capable of reaching into the rack for the second load.

- Double deep storage with double-load-width aisles. The S/R machine carries two loads at a time and inserts them simultaneously into the double deep cubicle.
- Deep lane storage with single-load-width aisles. An S/R machine dedicated to *storing* will store material into the lanes on either side of the aisle. The lanes may hold up to 10 loads each. On the output side, a dedicated *retrieval* machine will remove material from the racks. The racks may be dynamic, having gravity or powered conveyor lanes.
- Rack entry module (REM) systems in which a REM moves into the rack system and places/receives loads onto/from special rails in the rack (see Fig. 8).

Another variation of the typical configuration is the use of transfer cars to transport S/R machines between aisles. Transfer cars are used when the storage requirement is high relative to the throughput requirement. In such a case, the throughput requirement does not justify the purchase of an S/R machine for each aisle, yet the number of aisles of storage must be sufficient to accommodate the storage requirement.

A third system variation is the number and location of I/O points. Throughput requirements or facility design constraints may mandate multiple I/O points at locations other than the lower left hand corner of the rack. Multiple I/O points might be used to separate inbound and outbound loads and/or to provide additional throughput capacity. Alternative I/O locations include the top of the system at the end of the rack (some AS/RS are built underground) and the middle of the rack.

In considering alternative configurations, the system designer should keep in mind that the more complex the system is, the more expensive it will be. These additional expenses may be justified by careful consideration of the throughput and storage requirements of the system, where variations from typical configurations are warranted based on extreme variety ratios (number of loads to be stored to the number of SKUs) and/or extreme turnover ratios (number of loads to be stored to the required throughput).

3.2 Automated Small Load Storage

Miniload AS/RS, microload AS/RS, and carousels comprise nearly all small load automated S/R applications today. The miniload AS/RS and carousel applications are described in Sections 3.21 and 3.22. The microload AS/RS is described in Section 4.1.

3.2.1 *Miniload AS/RS*

The miniload AS/RS is called such because it typically handles loads weighing less than 750 pounds (miniloads) and because it operates much like a unit load AS/RS. The major difference between the miniload AS/RS and the unit load AS/RS, other than the load size, is the function for which the systems are designed. More often than not, the miniload is designed to support an order-picking operation (see Figs. 9 and 10).

Order picking is performed by an operator at the end of each miniload aisle. The operator picks from trays that are automatically sequenced and transported to the pick station by the S/R machine. The trays come in a variety of sizes and may be divided into as many as 72 modules, each of which contains a single SKU (see Fig. 11).

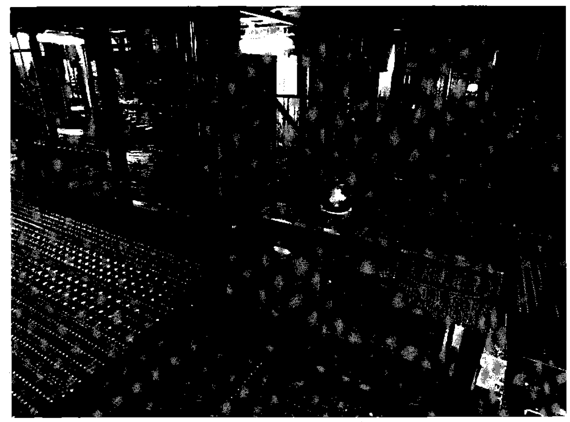


FIGURE 8 Inside a REM AS/RS (courtesy of Litton IAS).

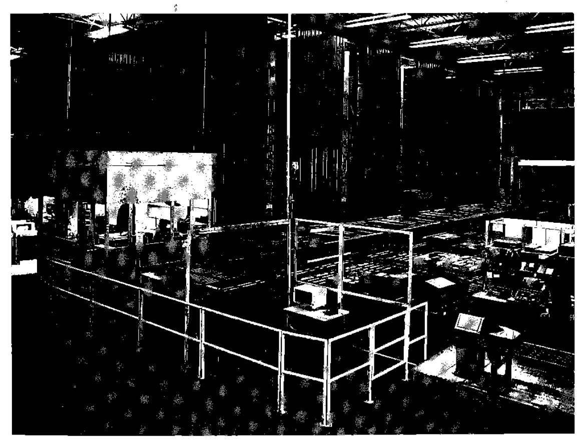


FIGURE 9 Typical miniload AS/RS with front end conveyor (courtesy of Eaton-Kenway Corp.).

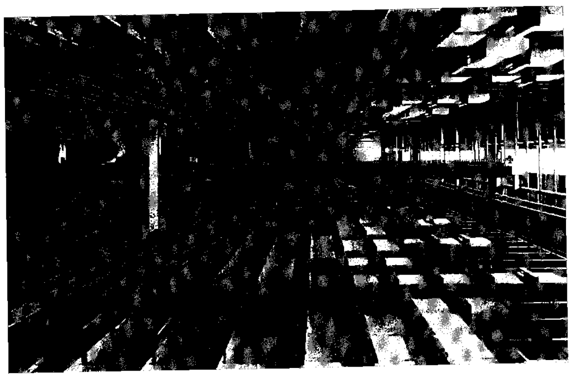


FIGURE 10 Inside a miniload AS/RS (courtesy of Litton IAS).

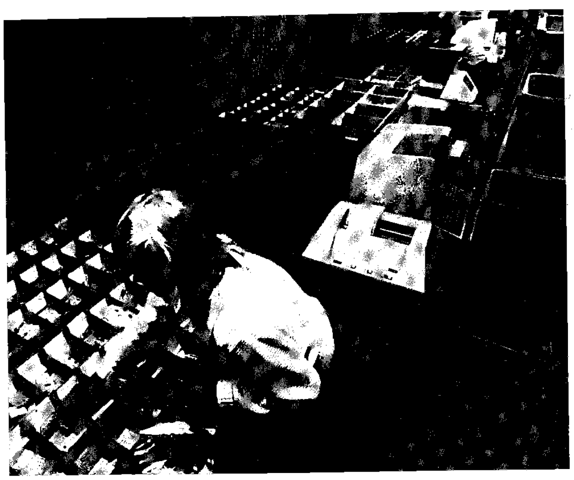


FIGURE 11 Order picking at a miniload AS/RS (courtesy of Litton IAS).

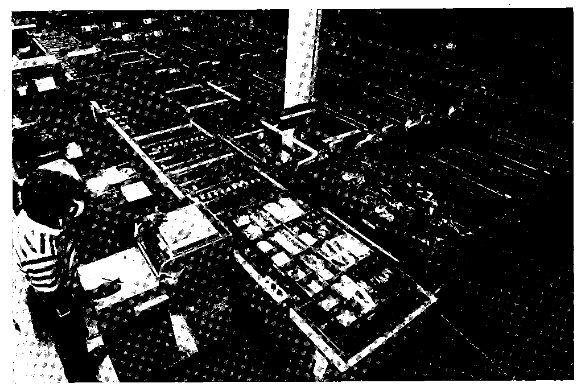


FIGURE 12 Miniload front end conveyor (courtesy of Litton IAS).

FIGURE 13 Typical horizontal carousel S/RS (courtesy of White Storage and Retrieval Systems).

As is the case with the unit load AS/RS, several alternative miniload configurations are available. Alternative configurations include multiple I/O points and I/O point locations, alternative front end configurations, and alternative ratios for the number of pickers to the number of aisles. If throughput is low relative to the number of items in the system, a picker may be assigned to more than one aisle. If throughput is high relative to the number of items in the system, more pickers than aisles may be required. In this case, the conveyor loop delivery system depicted in Figure 12 may be used to feed "remote" order pickers from the miniload.

3.2.2 Carousel Storage Systems

The variety of carousel types and applications has increased substantially in the last 5 years. Applications now include order picking, work-in-process storage, toolroom storage, kit storage, progressive assembly, and burn-in. Carousel varieties include horizontal, vertical, and independent rotating racks. Options for the material handling interface include human operators and robots.

A horizontal carousel is a series of linked bins mounted on an elongated track (see Fig. 13). When activated, the bins revolve, bringing the desired bin to the pick position. A horizontal carousel is distinguished from a vertical carousel by the fact that bins stand perpendicular to the floor and rotate around an axis perpendicular to the floor. Horizontal carousels are typically 30 to 50 feet long and 6 to 25 feet tall. Restrictions on system length stem from throughput requirements. The longer the carousel is, the more time it will take to present the operator/robot with the desired bin. Height restrictions are imposed by the reaching height of a human.

Vertical carousels (see Fig. 14) are distinguished from horizontal carousels by several key features. First, bins are aligned horizontally and revolve about an axis parallel to the floor. Second, the sesired bin, and only the desired bin, is always presented at *waist* height to the picker. Third, the carousel is enclosed by sheet metal, thus improving item security and protecting items from the environment. Fourth, vertical carousels typically are taller than horizontal carousels, in the range of 20 to 50 feet. Consequently, for equivalent storage requirments, they occupy less *square* footage than a horizontal carousel. Finally, vertical carousels are usually more expensive per *cubic* foot of storage space than horizontal carousels.

A third variety of carousel is the independent rotating rack carousel (see Fig. 15). In this configuration, each level is driven separately, permitting independent, simultaneous rotation. The significantly greater cost of this system is justified by greater throughput, because the picker (human or robot) spends less time waiting for rack rotation to position the proper bin.

Both humans and robots can serve as the I/O interface to carousel S/RS. Robots can be used (see Fig. 16) when the items to be retrieved from the carousel are uniformly sized and shaped, such as a tote pan. Robots remove the system height restrictions and load weight restrictions imposed by humans.

3.3 Microcomputers in Storage System Control

Automated storage systems typically employ a three-level hierarchical control architecture. At the lowest level is the control of motors, activators, sensors, and other physical devices. This level typically will employ a programmable

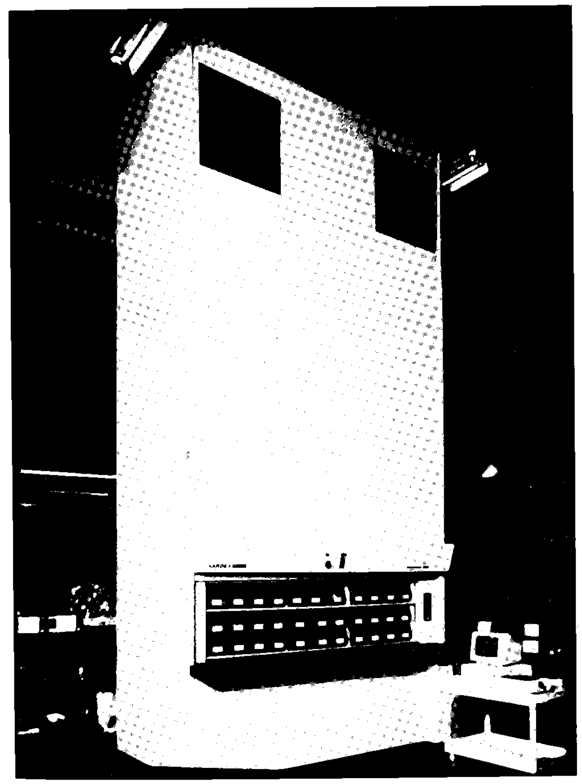


FIGURE 14 A vertical carousel (courtesy of Kardex).

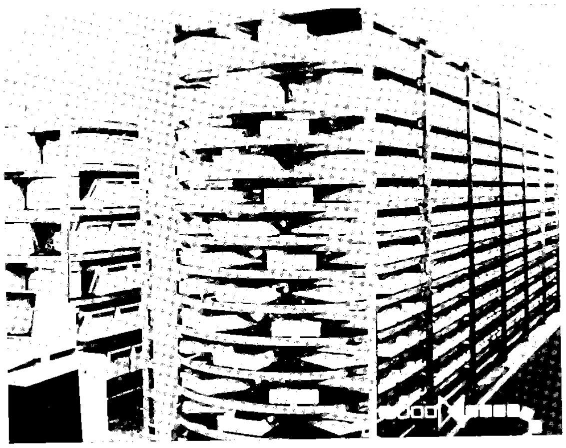


FIGURE 15 Independent rotating rack carousel (courtesy of Nestier).

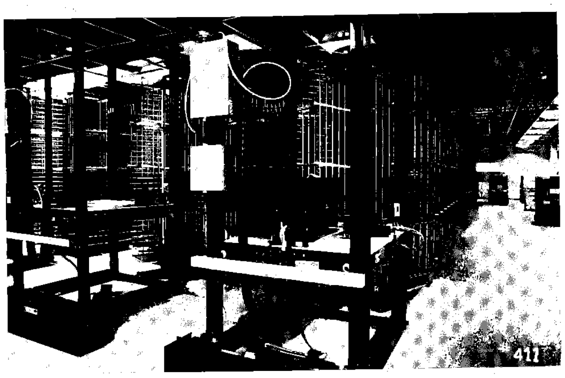


FIGURE 16 Robot interface to a horizontal carousel (courtesy of White Storage and Retrieval Systems).

controller (PC) or programmable logic controller (PLC), a type of microcomputer. The second level in the hierarchy is the "aisle controller," which directs a single S/R machine or carousel. The aisle controller often will be a microcomputer, with an operating system, executing programs written in a high level language. The aisle controller "manages" the storage system to complete a set of S/R operations most efficiently. It communicates with the S/R or carousel controller using a very limited vocabulary:

"go to rack 12, level 3, and extract a load,"
"move from rack 9, level 4, to rack 6, level 5,"
"unexpected empty location,"
"unexpected load encountered,"
etc.

The aisle controller must maintain a "map" of the storage locations, including the load status and load identity for each location. It must incorporate some method for arbitrating competing service requests, for example, a load to be stored and several unfilled retrieval requests.

The nature of the aisle controller depends significantly on the storage system type and application. For example, in a warehousing AS/RS, it may be sufficient to distinguish loads to be stored as either fast or slow movers, whereas for the order-picking miniload, it may be important to know which SKUs are often picked together with the one to be stored.

The third level in the hierarchy is the area host, which may be directing several aisles or carousels. The area host will be originating retrieval requests and may be distributing items for storage between the aisles. The area host generally will be a minicómputer or a program running on a mainframe.

The PC or PLC will receive high-level instruction from the upper levels of the hierarchy and will return completion/error codes. It is the key link in the interface between the digital world of computer control systems and the analog world of physical devices.

3.4 Automated Storage System Design

Automated storage systems require significant capital and design committments. A unit load AS/RS, miniload AS/RS, or a carousel S/RS can cost millions of dollars. In addition, automated storage systems are not easily reconfigured or liquidated if they are poorly designed. As a result, it is essential that the system be engineered to meet targeted storage and throughput requirements for the planning horizon under consideration.

The engineering design problem can be formulated as follows:

Objective: Minimize Cost
Subject to: Storage requirements
Throughput requirements
Building restrictions

For an AS/RS, the solution to this problem usually means minimizing the number of aisles that will satisfy storage requirements, throughput requirements, and building restrictions. Building restrictions and throughput requirements are relatively easy to define. However, storage requirements depend on the type of storage policy in place.

Once the requirements have been defined, checking whether a given design satisfies storage requirements and building restrictions is straightforward. However, checking for satisfaction of throughput requirements requires the application of analytically developed expected cycle time expressions and/or simulation.

For carousel systems the process is similar except that the minimization is over the number of carousels as opposed to the number of aisles.

The microcomputer is a very efficient tool for implementing this design algorithm.

Once a storage system has been designed and installed, changing circumstances will inevitably require improvements in throughput capacity and space utilization. Throughput can be improved by clever assignment of items to locations, such as assigning high-activity items closer to the front of the system and assigning items that are likely to be requested together in the same storage location. Space utilization can be improved through a randomized storage policy and by assigning items to appropriately sized locations. Again, the microcomputer is an effective tool for executing these strategies.

4.0 INTEGRATED STORAGE AND TRANSPORT

Most material handling applications involve both transport and storage of material. Thus, it is not surprising to find automated material handling systems that integrate storage and transport functions. Integrated storage and transport systems, or ISTS, involve more than simply automated storage and automated transport, as described earlier. ISTS require both a physical interface and an "intelligent" mechanism for coordinating the control of the storage and transport functions.

Sections 4.1 and 4.2 provide brief descriptions of two approaches to ISTS. These certainly are not the only approaches, and the interested reader should consult the references cited at the end of the article, as well as the trade press.

4.1 Microload AS/RS

One increasingly popular approach to ISTS is really an evolutionary development of the control system for unit load AS/RS. The microload AS/RS is a scaled-down version of the unit load AS/RS, with typical load capacities in the range of 75 to 300 pounds. Figure 17 illustrates a microload AS/RS application in electronics assembly.

Where the microload AS/RS differs most significantly from its unit load predecessor is in its ability to identify and track the unit loads. The S/R machine incorporates a bar code scanner, which is used to identify the unit loads by their distinctive bar code labels. This allows workstations located along the outside of the rack structure to receive containers of material from an "input" location and, subsequently, to deliver completed jobs to an "output" location. Thus, a unit load may be deposited by the S/R machine into a "storage" location, which is a workstation input port, and later picked up by the S/R machine from another "storage" location, which is the workstation output port. If the S/R machine could not obtain a positive identification of the unit load, then some form of intervention would be required to determine the disposition of the unit load.

The microload AS/RS is capable of providing both the required automated storage and the automated movement of material between workstations. This technology has found applications in a range of industries, and several vendors offer products in this equipment category. The fundamental structure of the microload AS/RS can accommodate variations in the container design, for example, from tote pans to specialized pallets that permit "hot" or power-on testing of electronic assemblies. The control systems can be customized for complex container routing, for combinations of subassembly containers and component part containers, and for a variety of operations, including both manufacturing processes and inventory control operations such as cycle counting.

One of the appealing aspects of the microload AS/RS approach is that a highly automated cell based on this technology could operate as an "island of automation," that is, it could be operated without any direct link to other computer systems or other automated material handling systems. Thus, it could be implemented without having to address the complex issues of communication between different pieces of automated equipment or different computers.

4.2 Carousel/Transporter Systems

Another popular approach to ISTS, especially for light assembly, is the combination of carousels for centralized storage and transporter conveyors for material movement. The physical interface between the two automated systems may be provided by an operator or by a robotic insertion/extraction device. Figure 16 shows an example of an automated interface.

As with the microload AS/RS, there is a requirement for positive identification of the unit loads at some point in the handling system. One alternative

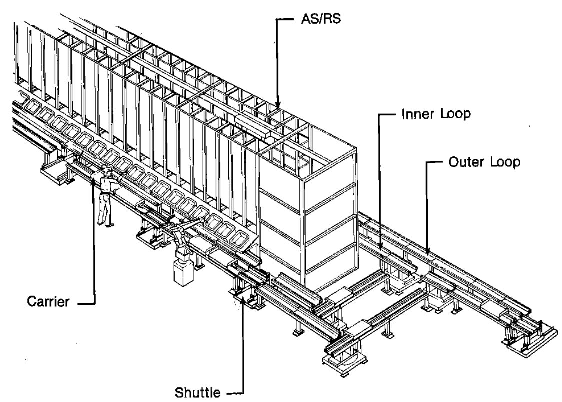


FIGURE 17 Microload AS/RS (courtesy of SI Handling Systems, Inc.).

is to read a bar code on the containers as they are presented for storage. Subsequently, the end-of-aisle controller for the carousel will maintain the information about the storage location, so that the container may be withdrawn as required by the production activity control system. The correspondence between container label and container content must be maintained either by the end-of-aisle controller or by the cell host.

Control of a carousel/transporter system may be more complex than for the microload AS/RS, simply because there are two or three distinct pieces of automated equipment, instead of just one. As a result, there is a requirement for communication and message-level- integration. Although there are emerging standards for communication, the message-level integration is still a significant problem.

5.0 AUTOMATED CONTROL

Automated control includes automated physical and status control of material. Physical control is the control of the orientation of, sequence of, and space between material. True automation of this function typically requires robotic handling. Status control is the real-time awareness of the location, amount, destination, origin, owner, and schedule of the material. True automation of this function requires automatic identification.

5.1 Robotics and Material Handling

A robot is defined by the Robotic Industries Association to be a "reprogrammable, multifunctional, manipulator designed to move material, parts, tools, or specialized devices through variable programmed motions for the performance of a variety of tasks." Others describe a robot as an automatic device that performs functions ordinarily ascribed to human beings and that operates with what appears to be almost human intelligence and perception. Figures 18 through 21 illustrate some typical robot applications.

As of the end of 1985, approximately 20,000 such devices, at a value of over $1 billion, were installed in the United States. Of those 20,000, nearly half were devoted to material handling applications.

Material handling applications for robotics include loading/unloading machine tools, palletizing and/or packaging, and small parts flexible assembly. Applications are especially attractive in hazardous and uncomfortable environments, where operations are highly repetitive, where load sizes are fairly uniform, and where high reliability and repeatability of motions are required. With this growing list of applications and functions, it is no wonder that the market for material handling robots is expected to grow at a rate of 20% to 30% over the next several years.

As the applications for robotics have increased, so has the variety of robot "personalities." A robot's "personality" can be expressed by the way it is powered, controlled, configured, and by its placement, mobility, and price.

Robots can be powered hydraulically, electrically, or pneumatically. Electric power is usually cleaner but less powerful than hydraulic.

Robot control is classified as servo or nonservo. Servo-controlled robots have the motion of their arm constantly controlled as it moves through a

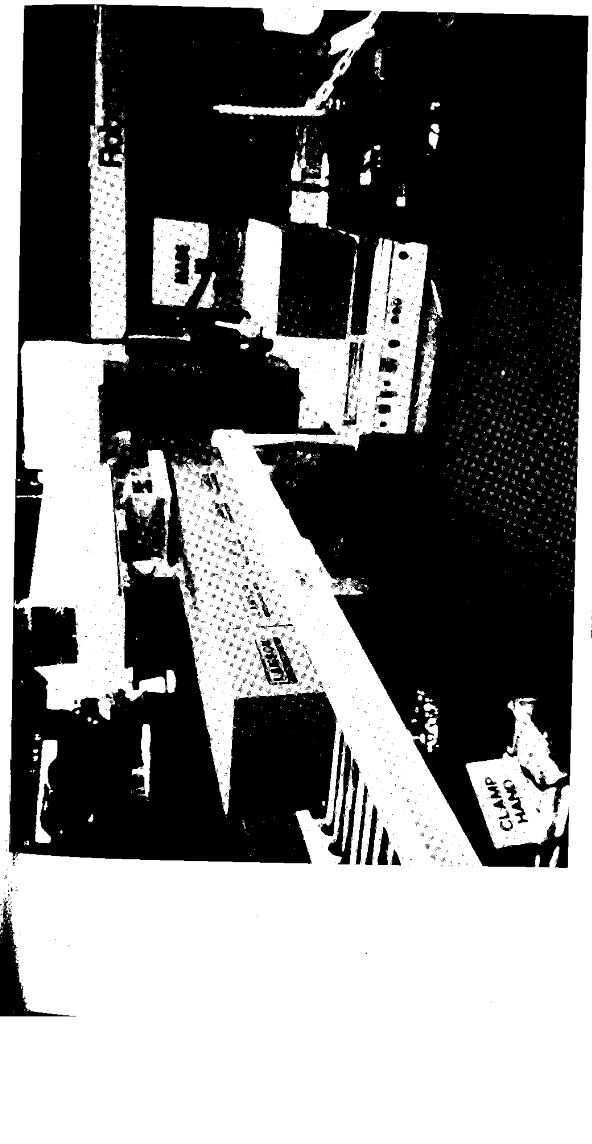


FIGURE 18 Robotic paletizer (courtesy of Robopal).

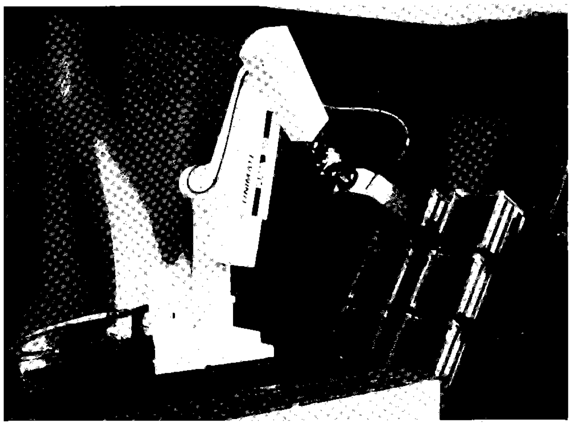


FIGURE 19 Robot load/unload station (courtesy of Unimation).

desired path. Control is performed by a computer, which receives feedback relating to arm position. Servo-controlled robots are generally more capable and therefore more popular than nonservo varieties. Servo-controlled robots are also more expensive.

Nonservo-controlled robots use a mechanical stop to control placement of the robot arm. The number of stops that can be built into the robot is limited. Consequently, nonservo-controlled robots are typically used in pick-and-place applications.

Robot configurations include rectangular (or Cartesian), spherical (or polar), cylindrical, jointed arm, and selective compliance assembly robot arm (SCARA).

Robots can be mounted on the floor, ceiling, or wall. A robot may be stationary, track mounted, or mobile on an AGV. Mobile robots are one of the most promising application areas for robotics in the future.

Finally, robot prices range from \$20,000, for a simple rectangular, electric, pick-and-place, nonservo-controlled robot, to \$100,000 plus, for a sophisticated, cylindrical, hydraulic, assembly, servo-controlled robot.

5.2 Automatic Identification

Automated status control of material requires that the real-time awareness of the location, amount, origin, destination, and schedule of material be achieved automatically. This objective is in fact the function of automatic identification technologies, technologies that permit real-time, nearly flawless data collection. Examples of automatic identification technologies at work include

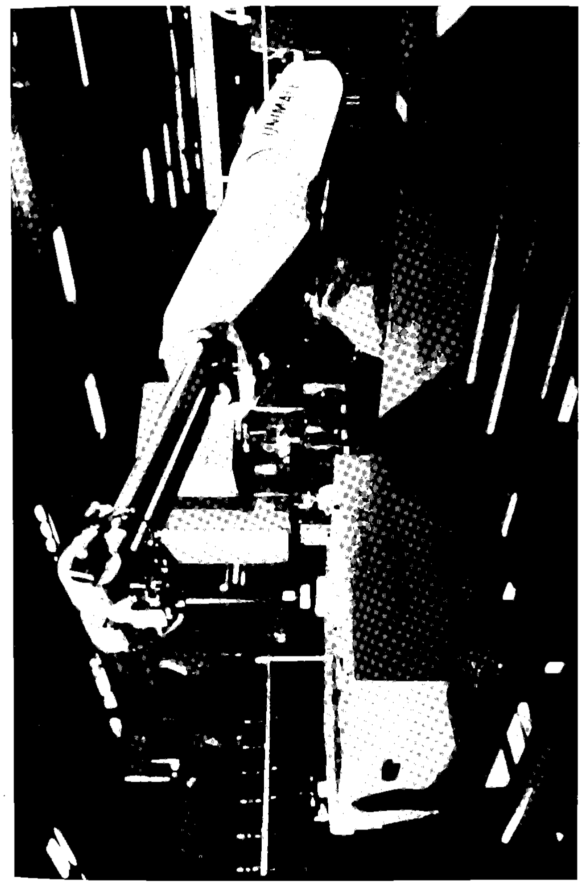


FIGURE 20 Robotic unitizer (courtesy of Unimation).

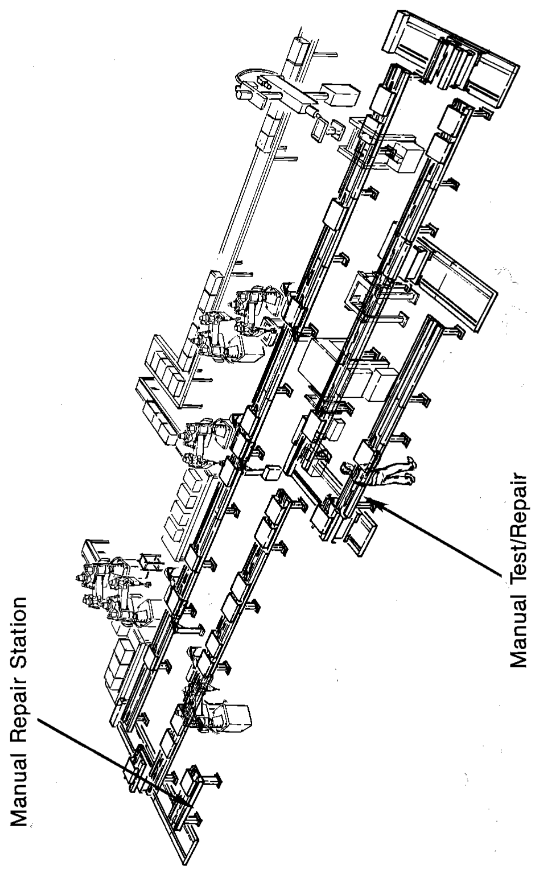


FIGURE 21 Robots at work in a flexible manufacturing system (courtesy of SI Handling, Inc.).

- A vision system reading bar code labels to identify the proper destination for a carton traveling on a sortation conveyor.
- A laser scanner to relay the inventory levels of a small parts warehouse to a computer via RF.
- A voice recognition system to identify parts received at the receiving dock.
- A RF or surface accoustical wave (SAW) tag used to permanently identify a tote pan.
- A card with a magnetic stripe that travels with a unit load to identify the load through the distribution channels.

Each of these technologies is described below in Sections 5.2.1 through 5.2.6.

5.2.1 Bar Codes and Bar Code Readers

The most widely used technology for automatic identification is bar coding. A bar code consists of a number of printed bars and intervening spaces (see Fig. 22). The structure of unique bar/space patterns represents various alphanumeric characters. The same pattern may represent different alphanumeric characters in different codes.

Bar codes are read by both contact and noncontact scanners. Contact scanners can be portable or stationary and use a wand or a light pen. The wand/pen is manually passed across the bar code. The scanner emits either white or infrared light from the wand/pen tip and reads the light pattern that is reflected from the bar code. This information is stored in solid-state memory for subsequent transmission to a computer (see Figs. 23 and 24).

Contact readers are excellent substitutes for keyboard or manual data entry. Alphanumeric information is processed at a rate of up to 50 inches per minute, and the error rate for a basic scanner connected to its decoder is 1 in 1,000,000 reads.

Light pen or wand scanners with decoder and interface cost around $2,500.

Noncontact readers are usually stationary and include fixed-beam, moving-beam scanners, and charged couple device (CCD) scanners.

Fixed-beam readers use a stationary light source to scan a bar code. They depend on the motion of the object to be scanned to move past the beam. Fixed-beam readers rely on consistent, accurate code placement on the moving object.

Moving-beam scanners employ a moving light source to search for codes on moving objects. Because the beam of light is moving, the placement of the code on the object is not critical. In fact, some scanners can read codes accurately on items moving at a speed of 1,000 feet per minute.

CCD scanners have only recently been used to read bar codes. A CCD scanner is more like a camera than a bar code scanner. By changing camera lenses and focal lenghts, the scanners read various bar codes at various distances. Like any camera system, the quality of the picture (the accuracy of the read) depends on the available light. When installed correctly, the CCD scanner is 99% accurate and costs around $4,000, including decoder.

5.2.2 Optical Character Recognition

Optical character recognition (OCR) systems read alphanumeric data so that people as well as computers can interpret the information. The OCR label is

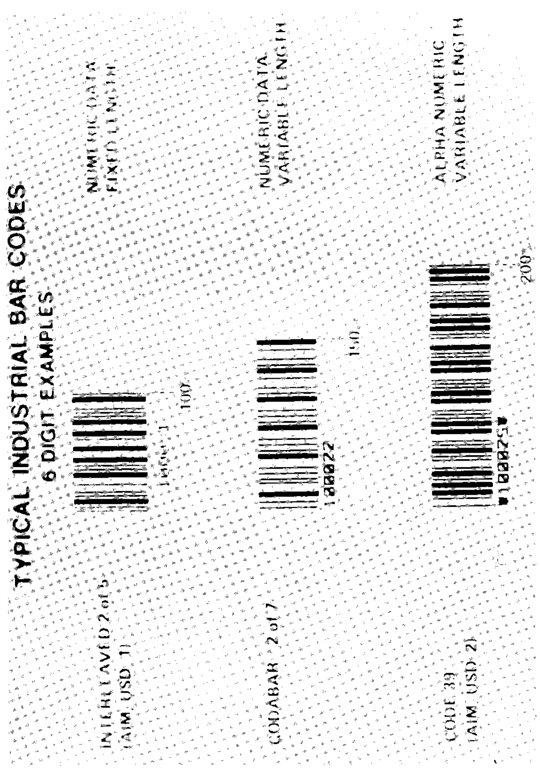


FIGURE 22 Typical industrial bar codes.

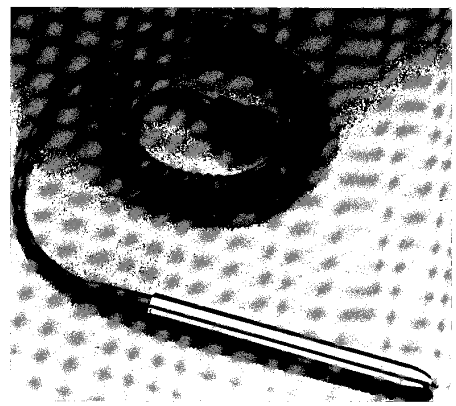


FIGURE 23 A Light Pen Bar Code Reader (courtesy of Computer Identics).

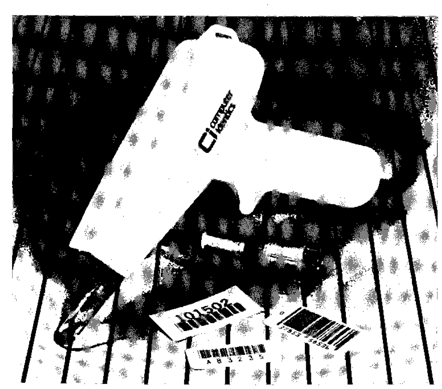


FIGURE 24 Hand-held laser scanner (courtesy of Computer Identics).

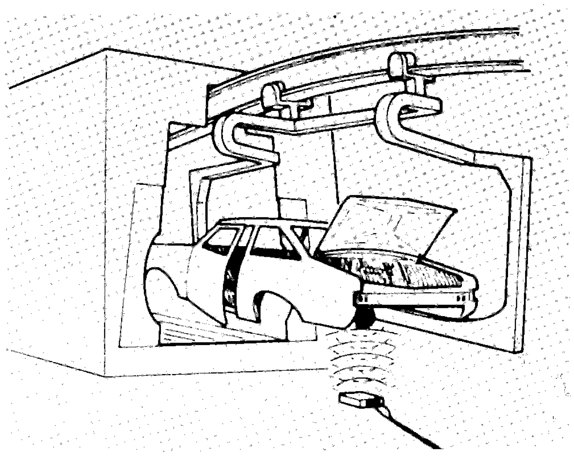


FIGURE 25 Automatic identification by RF.

read with a hand-held scanner, much like a bar code. OCR systems operate at slower read rates than bar code systems and are priced about the same. OCR systems are attractive when both human- and machine-readable capabilities are required. Optical characters are found at the bottom of bank checks.

5.2.3 *RF and SAW*

Both RF and SAW techniques encode data on a chip that is encased in a tag. When a tag is within range of a special antenna, the encased chip is decoded by a tag reader. RF tags can be programmable or permanently coded and can be read from up to 30 feet away. SAW tags are permanently coded and can be read only within a 6-foot range (see Fig. 25).

RF and SAW tags are typically used for permanent identification of a container, where advantage can be taken of the tag's durability. RF and SAW technologies are also attractive in harsh environments where printed codes may deteriorate and become illegible.

A tag reader costs around $10,000. Nonprogrammable tags range from $8 to $50; programmable tags, from $50 to $150.

5.2.4 *Magnetic Stripes*

Magnetic stripes are used to store a large quantity of information in a small space. The stripe is readable through dirt or grease, and the data contained in the stripe can be changed. The stripe must be read by contact, thus eliminating high-speed sortation applications.

Magnetic stripe systems are generally more expensive than bar code systems.

5.2.5 *Voice Recognition*

Voice recognition (VR) is a computer-based system that translates spoken words into computer data without special codes. VR systems are attractive when an operator's hands and eyes must be freed up for productive operations. Though VR systems are in their infancy, some systems recognize up to 1,000 words and are 99.5% accurate. VR systems are still relatively expensive and must be dedicated to one operator at a time.

5.2.6 *Vision Systems*

Vision system cameras take pictures of objects and codes and send the pictures to a computer for interpretation. Vision systems "read" at moderate speeds, with excellent accuracy at least for limited environments. Obviously, these systems do not require contact with the object or code. However, the accuracy of a read is strictly dependent on the quality of light. Vision systems are becoming less costly but are still relatively expensive.

6.0 CONCLUSION

Material handling has been defined as "moving, storing, and controlling material." When material handling is automated, all three of the basic functions depend heavily on microcomputers. In the form of programmable controllers, they provide the interface between the digital world in which data are analyzed and logical decisions are made, and the analog world in which data are obtained and decisions are implemented. In addition, microcomputers provide an interface between the programmable controllers driving physical devices and the higher-level control computers. In the future, microcomputers should begin to take over some of the higher-level control functions.

FOR MORE INFORMATION

Automated material handling encompasses a wide and growing range of equipment, computer, and communication technology. This article has touched briefly on the most common equipment types. For additional and up-to-date information, the interested reader is directed to two relevant periodicals:

Material Handling Engineering
Penton/IPC, Inc.
1111 Chester Avenue
Cleveland, OH 44114

Modern Materials Handling
Cahners Publishing Co.
275 Washington Street
Newton, MA 02158

The major trade association for manufacturers of material handling equipment and material handling systems integrators is

The Material Handling Institute, Inc.
8720 Red Oak Blvd.
Suite 201
Charlotte, NC 28210

The Material Handling Institute sponsors several trade shows and educational programs related to material handling. In addition, the Institute publishes a number of excellent booklets, videotapes, and slide sets on material handling.

The major academic research center for material handling is

The Material Handling Research Center
765 Ferst Dr.
The Georgia Institute of Technology
Atlanta, GA 30332-0205

In addition to conducting research on material handling problems in manufacturing and warehousing, the Material Handling Research Center sponsors two annual week-long educational programs focused on material handling.

The major professional society representing users of material handling equipment is

The International Material Management Society
8720 Red Oak Blvd.
Charlotte, NC 28210

BIBLIOGRAPHY

"A Status Report on Industrial Computer Networks," *Plant Eng.*, pp. 56–84 (September 12, 1985).
"A U.S. First—AGVs Move onto the Assembly Lines," *Mod. Mater. Handl.*, pp. 78–83 (January 1985).
Adams, Russ, "Warehouse and Bar Code," *Bar Code News*, pp. 14–17 (March/April 1986).
Andel, Tom, "Aerospace Manufacturer Unveils Flexible Manufacturing Components," *Mater. Handl. Eng.*, pp. 48–54 (May 1984).
Andel, Tom, "On-Board Computer Terminals Make for a Self-Correcting Distribution Ssytem," *Mater. Handl. Eng.*, pp. 109–113 (November 1984).
Asfahl, C. Ray, *Robots and Manufacturing Automation*, John Wiley, New York, 1985.

"Automated Storage Systems," *Mod. Mater. Handl.*, pp. 44–51 (October 1983).
"Automatic Guided Vehicles Move into the Assembly Line," *Mod. Mater. Handl.*, pp. 78–83 (January 1985).
"Automatic I.D. Delivers the Data for Warehouse Efficiency," in *Mod. Mater. Handl. 1986 Warehousing Guidebook*, pp. (77–80).
"Automatic Identification," *Mod. Mater. Handl.*, Supplement, pp. 98–105 (March 5, 1984)
Automatic Identification Systems for Material Handling and Material Management. Material Handling Institute, Charlotte, NC, 1976.
"Automating Appliance Manufacturing," *Manuf. Eng.*, p. 61 (August 1983).
Behne, Thomas A., "Opel Engine Line Goes Flexible," *Automot. Ind.*, pp. 23–24 (August 1984).
Bergstrom, Robin P., "Automated Pump Assembly Pays Off," *Manuf. Eng.*, pp. 44–47 (June 1984).
Binning, Ronald L., "New Uses of Traditional Storage Systems Can Minimize Inventory And Work-in-Process," *Ind. Eng.*, pp. 81–83 (March 1984).
Budill, Edward J. "AS/RS Capabilities and Design Considerations," A presentation to the Material Hanlding Institute Teachers Conference, 1983.
"Casebook," *Mod. Mater. Handl.*, Casebook Issue, 39(15) (1985).
"The Coming of the Automatic Factory," *Manuf. Eng.*, pp. 69–75 (March 1980).
"Computer-Aided Order Picking: Paperless Picking That's Accurate and Very Fast," *Mod. Mater. Handl.*, pp. 44–47 (September 6, 1983).
"Computer-Integrated Manufacturing Systems," *Mod. Mater. Handl.*, pp. 96 (May 1985).
"Computer-Run Cars Make Flexible Storage Work," *Mod. Mater. Handl.*, pp. 83–85 (March 1985).
"Conference Speakers Tout Technology Advances," *Mod. Mater. Handl.*, pp. 68–70 (August 1985).
"Control Key for Harris FMS," *Manuf. Eng.*, 91(3), 60 (September 1983).
"Conveyor Systems for Flexible Assembly Operations," *Mod. Mater. Handl.*, pp. 67–70 (July 1985).
Curtin, Frank T, "Automating Existing Facilities: GE Modernizes Dishwater, Transportation Equipment Plants," *Ind. Eng.*, pp. 32–38 (September 1983).
Dallas, Daniel B., "The Robot Enters the System," *Manuf. Eng.*, pp. 70–73 (February 1979).
Drozda, Thomas J., "Flexible Assembly System Features Automatic Setup," *Manuf. Eng.*, pp. 75–76 (December 1984).
Engwall, Richard L., "Automated Material Handling in Electronics Assembly," in *Proceedings of the Annual Material Handling Users Conference*, Georgia Institute of Technology, Atlanta, GA, September 1985.
Filley, Richard D., "Bar Code Guide and Survey Detail What's Available, How It Can Smooth Your Operation." *Ind. Eng.*, pp. 74–94 (September 1983).
"Flexible Assembly—A Boon for Short Production Runs," *Mod. Mater. Handl.*, pp. 48–54 (June 1984).
"Flexible Manufacturing Cuts Costs, Improves Performance," *Mod. Mater. Handl.*, 70–74 (September 1985).
"Flexible Manufacturing Systems—Handling's Critical Role," *Mod. Mater. Handl.*, pp. 58–70 (September 1982).

Foulkes, Fred K., and Jeffrey L. Hirsch, "People Make Robots Work," *Harv. Bus. Rev.*, pp. 94–102 (January–February 1984).
Francis, R. L., and J. A. White, *Facility Layout and Location: An Analytic Approach*, John Wiley, New York, 1975.
Frazelle, Edward H., "Design Problems in Automated Warehousing," in *Proceedings of the 1986 IEEE Conference on Robotics and Automation*, San Francisco, CA, April 8, 1986.
Frazelle, Edward H., "Material Handling: A Technology for Industrial Competitiveness," Material Handling Research Center Technical Report, Atlanta, GA, April 25, 1986.
Higgins, William J., "Robots in Manufacturing," in *Machine and Tool Blue Book*, June 1984, pp. 46–55.
Hill, John, "Automatic Identification And Sortation Systems Capabilities and Design Considerations," in *Automated Material Handling and Storage*, (J. A. Tompkins and J. D. Smith, eds.), Auerbach Publishers, Pennsauken, NJ, 1983.
Hill, John, "Will Automatic Identification by Radio Frequency Solve Your Material Handling Problem?" *Mat. Handl. Eng.*, pp. 100–102 (November 1984).
Horrey, R. J., "Sortation Systems: From Push to High-Speed Fully Automated Applications," *Proceedings of the Fifth International Conference on Automation in Warehousing* (ICAW), Atlanta, GA, Dec. 4–7, 1983, pp. 77–83.
"IBM's Automated Factory—A Giant Step Forward," *Mod. Mater. Handl.*, pp. 58–65 (March 1985).
"Industrial Robots," *Mod. Mater. Handl.*, pp. 46–53 (March 7, 1980).
"Industrial Trucks—Why Today's Designs Give You More for Your Operating Dollar," *Mod. Mater. Handl.*, pp. 84–89 (January 1985).
Kehoe, Louise, "The 27-Second Big Mac Attack," *Electron. Bus.* (April 15, 1984).
Kerr, John, "Priam Stays Onshore with Automated Factory," *Electron. Bus.* (July 1984).
Knill, Bernie, "Vought Aero Products Reveals New Lessons in Flexible Manufacturing," *Mat. Handl. Eng.*, pp. 78–85 (January 1985).
Knill, Bernie, "Ford Takes a New Material Handling Route to Board Testing," *Mater. Handl. Eng.*, pp. 76–80 (September 1985).
Kulwiec, Ray (ed.), *Materials Handling Handbook*, John Wiley, New York, 1985.
Lofgren, Gunnar, "Automatic Guided Vehicles Perform as Production Line Systems," *Ind. Eng.*, pp. 36–38 (November 1981).
"The Major Focus Is on Integrated Operations," *Mod. Mater. Handl.*, pp. 66–71 (January 1985).
Mariotti, John J., "Assembly Line Design: Choosing And Setting Up Conveyor Systems," *Ind. Eng.*, pp. 52–56 (August 1981).
"Material Handling: A Technology of Productivity," *Business Week*, Supplement (January 21, 1985).
"Mazak's FMS: Making Machine Tool Manufacturing Look Easy," *Manuf. Eng.*, p. 63 (September 1983).
"Mini-load AS/RS trims inventory, speeds assembly," *Mod. Mater. Handl.*, pp. 47–49 (September 1984).
Nagy, Bence A., "A Flexible Assembly System for Printer Production," in

Proceedings of the Annual Material Handling Users Conference, Georgia Institute of Technology, Atlanta, GA, September 1985.
Nof, Shimon (ed.), *The Handbook of Industrial Robotics*, John Wiley, New York, 1985.
Orr, Gary B., Scott M. Sopher, and James M. Apple, "Material Handling Equipment Alternatives Examined for Progressive Build in Light Assembly Operations," *Ind. Eng.*, pp. 68–73 (April 1985).
"Our Flexible Systems Help Us Make a Better Product," *Mod. Mater. Handl.*, pp. 58–62 (April 1985).
Pierson, Rober A., "Adapting Horizontal Material Handling Systems to Flexible Manufacturing Setups," *Ind. Eng.*, pp. 62–71 (March 1984).
"Planning Dismantles the Barriers" Networks of Flexible Flow," *Prod. Eng.*, pp. 44–48 (March 1984).
Podolsky, Doug M., "The ITC Assesses the U.S. Robotics Industry," *Robotics World*, pp. 8–10 (February 1985).
"Productivity in Manufacturing: Some Equipment You'll Encounter," *Mater. Handl. Eng.*, pp. 100–114 (January 1985).
Quinlan, Joe, "Rotating for Profit," *Mater. Handl. Eng.*, pp. 88–91 (October 1980).
Radio Terminals Improve Warehouse Productivity," *Mod. Mater. Handl.*, pp. 60–61 (August 1985).
Rhea, Nolan W., "Miniloads, Carousels Offer Big Benefits to Small-Load Handling," *Mat. Handl. Eng.*, pp. 42–47 (November 1983).
Rhea, Nolan, "Unit Load AS/RS Meets Just-in-Time Manufacturing," *Mater. Handl. Eng.*, pp. 58–62 (November 1984).
Riechenback, Raymond H., "Automated Typewriter and Printer Assembly," in *Proceedings of the Annual Material Handling Users Conference*, Georgia Institute of Technology, Atlanta, GA, September 1985.
"Robotized Servo Manufacture and Assembly," *Manuf. Eng.*, pp. 53–55 (June 1983).
Robots: Tireless Players on the Manufacturing Team," *Mod. Mater. Handl. 1985 Manufacturing Guidebook*, pp. 64–65.
"The Role of Handling in Factory Automation," *Mod. Mater. Handl.*, pp. 51–57 (July 1983).
Scaringe, Robert A., "The Radio Frequency Control Links: A Mobile, Real-Time Data Collection Technology," *Ind. Eng.* (November 1985).
Schwind, Gene, "GM's Orion Assembly Plant: Another Step up the Automation Ladder," *Mater. Handl. Eng.*, pp. 36–41 (May 1984).
Schwind, Gene, "Chrysler Windsor: A Plant in a Hurry to Turn out Vans in a Hurry," *Mater. Handl. Eng.*, pp. 50–54 (November 1984).
Schwind, Gene, "Machine Vision: Eyes of the Automated Factory," *Mater. Handl. Eng.*, pp. 117–122 (May 1985).
Schwind, Gene, "AGVS for Assembly: Flexible Layout, Easy Expansion," *Mater. Handl. Eng.* (September 1985).
Soltis, David J., "Automatic Identification Systems: Strengths, Weaknesses, and Future Trends," *Ind. Eng.*, pp. 55–59 (November 1985).
"S/R Machine Delivers Directly to Machine Tools," *Mod. Mater. Handl.*, pp. 56–57 (February 1985).
"Superior Control Pays Off in Flexible Assebmly," *Mod. Mater. Handl.*, pp. 48–52 (August 1985).
Tompkins, James A., and Jerry D. Smith, "Keys to Developing Material

Handling System for Automated Factory are Listed," *Ind. Eng.*, pp. 48–54 (September 1983).
Tomplins, J. A., and J. A. White, *Facilities Planning*, John Wiley, New York, 1984.
"Vertical Carousels Boost Picking, Conserve Space," *Mod. Mater. Handl.*, pp. 86–87 (September 1985).
Warehouse Modernization and Layout Planning, NAVSUP Publication 529, March 1985.
"We Achieved Flexible Flow to 250 Assembly Stations," *Mod. Mater. Handl.*, pp. 48–51 (October 5, 1984).
"Where Materials Handling Robots Are Headed," *Mod. Mater. Handl.*, pp. 68–73 (May 1985).
"Where Robots Are Headed in Materials Handling," *Mod. Mater. Handl.*, pp. 68–73 (May 1985).
Williams, Ray, "Extrusion Trim Cell," in *Proceedings of the Annual Material Handling Users Conference*, Georgia Institute of Technology, Atlanta, GA September 1985.
Witt, Clyde, "Carousels Make Their Move in Material Handling Systems," *Mater. Handl. Eng.*, pp. 50–51 (December 1984).
Witt, Clyde E., "Self-Powered Monorail Carriers Add Handling Value to Body Paint Line," *Mat. Handl. Eng.*, pp. 60–64 (June 1985).

LEON F. MC GINNIS
EDWARD H. FRAZELLE

AUTOMATED OFFICE INFORMATION SYSTEMS DESIGN

1.0 INTRODUCTION

Automation of office work is one of the fastest growing application areas of information systems (IS); many research efforts have been conducted during the last few years to provide guidelines with systematic foundations and support tools for the design of office information systems (OIS) [*1–5*]. In developing new approaches to OIS design, identification and consideration of the different facets and original aspects of the office environment constitute a fundamental issue [*6, 7*]. These original features are relevant even in the early phases of the design process, commonly called the "conceptual" or "logical" phases, the aim of which is to make the OIS design process easier and more reliable, providing conceptual and practical guidance to the analyst in understanding an existing office and in designing the corresponding computer-assisted procedures.

In current research, the problem of formally specifying office elements in a model has been investigated in more depth than that of developing a complete methodological approach to OIS design. Moreover, office design methodologies in existence do not offer complete frameworks but, instead, emphasize only a subset of the OIS design phases. Active research projects for new methodologies and design support tools have been undertaken to make the introduction of new technologies in the office quicker and more effective.

In an office environment, tools for supporting the development process should provide facilities for requirements collection and analysis, logical design, architectural design, and rapid prototyping. These tools are computer-based instruments for collecting information, providing graphic interfaces to model the office in a formal way, supporting consistency checks, and evaluating the system being designed.

The purpose of this article is to review the main areas of interest of OIS design and to present some advanced results in the field.

First, special characteristics of OIS are presented in Section 2.0. A survey of the principal types of office models, with examples, is given in Section 3.0. In Section 4.0, a relevant sample of office design methodologies is presented. The TODOS development environment, which is currently under implementation in the framework of the European ESPRIT project, is discussed as an example of a complete methodology for OIS design. Section 5.0 illustrates some well-known approaches to requirements specification in software engineering from which useful experiences of OIS design can be derived. Finally, Section 6.0 presents existing conceptual design support environments.

2.0 OFFICE CHARACTERISTICS

OIS design methodologies differ from design methodologies for conventional IS, because the nature of OIS is different from that of conventional IS.

Besides providing functions similar to those provided by conventional IS, OIS have the purpose of more generally supporting office work.

Among the functions of an OIS are filtering the large amount of data available in the office and providing the office workers with the specific information they need to perform their tasks. This can be useful both, for instance, to discard uninteresting mail or to handle it automatically and to select the data that must be analyzed to make a decision.

In conventional offices, a great aid to office work is offered by the arrangement of papers, books, and notes on the desk and on office shelves. This allocation has the function of reminding the office worker of activities to be executed and of their different priorities [8]. In an automated office, where most of the information is hidden from the user by the system, the system must provide the necessary function of reminding. This function can be improved by the automatic scheduling of activities to perform and meetings provided by the system or, better, supported by it.

The procedures of an OIS can be classified into services automatically performed and activities performed to support the work done manually by workers at workstations. The conceptual models on which OIS design methodologies are based must be semantically rich and consider office elements with their particular features. The types of data used in conventional IS, such as character, string, and numeric, are not enough to describe office data. Other types of data are currently employed in an office and must be supported by an OIS. These are unstructured data, such as data contained in messages, letters, texts, annotations, graphics, and oral communications.

A paper, for instance, is not composed solely of its title and authors, but also of the text, and the structure of the text is poorly handled if reduced only to sections and paragraphs. From the examples given above it also appears that the support for storage and communication of information is different for different types of data representations; for example, we can use textual VDU displays, graphic displays, or voice synthesizers and recognizers.

Another aspect that is essential in the specification of office elements is the time factor. Time is needed to determine the lifetime of documents or single operations, or to specify activities durations and timing constraints, and it is used for scheduling activities execution, for calendar functions, and for performing control operations on the OIS. Data relating to time must be defined in the system, together with all the operators that can be applied to such data. Flexibility is also needed in the definition of time constraints, which are usually lax. For instance, a business letter must be answered within an approximate time interval.

Another characteristic of data in office elements is that data elements are most frequently used together in groups (such as in independently stored documents), instead of being manipulated as single pieces of information fitted together in a form; for instance, it is often useful to store a document such as a letter in its completeness.

Not only are data unstructured in OIS but also some of the actions that are performed on data. The same action can be performed in several ways, on the condition that the obtained result is the required one.

Most office activities, moreover, are performed once some form of instruction has been given to the worker that must execute them. This instruction is usually incomplete, and to complete the work, some knowledge is used that is only implied in the given instructions.

Flexibility in performing actions is essential for achieving office goals, because many exceptions are possible. Every exception that can arise during office work should be supported by the system, at least in a minimal way, for instance, it is necessary to determine what to do if a person needed for a certain decision is absent.

Complexity in an OIS is much higher than in a conventional IS designed for operational activities. There are many types of elements in each office, and these elements are related by several connections.

In general, the elements that are needed to execute office work are distributed among several office workers in the same office or different departments and can also be located externally to the office environment. Therefore, an important office function is communication among office workers and with the external world [9]. An OIS needs the mail function for supporting most of its activities.

Another aspect that must be considered in the analysis of OIS elements is the inherent high level of evolvability. The most unstable aspects of an office are the constraints defined on its elements, for instance, the authorization for a worker to execute an operation. Activities can also be changed in time, one of the possible reasons being that ways to process information change. Document types can also vary (even if with lower frequency), due, for instance, to new legal requirements.

A second characterizing feature of an OIS is the impact of technology. OIS technologies are not yet established and show fast evolution; hence, office methodologies should be as independent as possible of implementation details, particularly in the early conceptual phases of OIS design.

A third difference between OIS and conventional IS is related to usage characteristics. OIS are highly interactive. The user interfaces should be adequate for the type of work to be performed, and the system dialogue should be facile enough for the casual user.

In an OIS there is an integration of different functions, based on the cooperation among the elements in the office to achieve the goal (not always well defined) of an efficiently operating system. Typically, the work is done by different workers in various steps, different functions – communication, data processing, information manipulation and retrieval, personal assistance, and task management – are linked together and used by the same worker in a fast sequence, and the various functions often interrupt each other.

Another aspect of usage characteristics concerns the different roles that office workers play in using an OIS. OIS usage implies direct contact of office workers with the system; users tend to be nonprofessional (casual users) and diversified. This implies an effort to design a user-friendly system and requires the development of functions for high-level users (e.g., offering decision support facilities). Another implication is that some knowledge roles in the office are replaced by OIS.

In Sections 2.1 through 2.4, the characteristics of office tools, office data, office tasks, and "active" office systems, respectively, are examined in greater detail.

2.1 Office Tools

Some of the automatic tools for supporting office work that characterizes many office systems are discussed below: advanced interfaces, productivity tools, and information retrieval systems.

2.1.1 Interfaces

Advanced user interfaces are needed for OIS. Systems providing windows, image handling, and voice handling have been proposed [10–12].

Window managers now exist in many commercial systems used as workstations. They make interaction among activities easier, because it is possible to present different activities simultaneously on the screen. For instance, in VITRAIL [13], a prototype system developed within the KAYAK project, it is possible to run activities concurrently and follow their evolution on the screen.

The use of windows is usually connected with the use of advanced pointing devices (e.g., mouse) and the use of pop-up menus. Each window usually has its own menu. All workstations offer graphics facilities, and some of them (such as VITRAIL) also voice interfaces. Icons are often used to symbolize possible activities, instead of using written commands.

Advanced image modeling allows image display through facsimile or other image input devices, and creation of correct and duplicate documents with image and graphics, including scaling (enlarging and reducing) document images [11].

Document models have been studied for handling mixed-mode documents. According to Horak and Kronert [14] two different structures are associated with each document: a logical structure and a layout structure. Both are expressed hierarchically as trees, so objects in figures can be basic objects or composite objects. Objects can be linked using construction rules to compose new object types; properties of documents can be specified to retrieve automatically appropriate values or to have default layouts for them (e.g., putting a logo in the document in the appropriate place). These properties may also be context sensitive; relation rules specify how to link document values, extracting them from different instances of documents, or a document may have different layouts depending on the output device.

Voice interfaces have also been proposed when voice documents are modeled or vocal messages are appended to existing written documents. A speech filing system has been proposed [15], where voice messages may be composed, edited, sent, and received, using telephones as terminals. A filing system provides selective retrieval, statistics, grouping, and scanning functions for messages.

2.1.2 Productivity Tools

Productivity tools are designated as office work support instruments, such as word processors, graphics, spreadsheets, data bases, and electronic mail.

An important subclass of productivity tools comprises in-house publishing systems that allow the preparation and manipulation of high-quality documents with text, tables, formulas, pictures, and so forth. Integrated systems for document generation typically support an editor that allows the specification of document content, commands for formatting, and a formatter that interprets the specified commands and produces the desired layout of the document. A thorough discussion of editors and formatters is beyond the scope of this presentation. An excellent overview on the subject can be found in Ref. *16* for the interactive editors and in Ref. *17* for the formatters.

Apart from the case of in-house publishing systems, productivity tools are generally not integrated in practice. Excellent systems are available for providing each tool separately; in some systems, functions are combined, but problems still exist in providing high-quality tools and managing the use of such tools [18]. Two outstanding problems relating to quality of tools are user interfaces and integration. User interface problems have been mentioned in Section 2.1.1. The problem of tool integration becomes important when workers utilize more than one tool. To introduce tools in the organization, workers must be persuaded to use them effectively. This involves training, providing continuous support, and generating a good environment to encourage their use. It is therefore necessary to include models of these tools, with the type of data they support and their functionality also in a conceptual model of the office in order to be able to include them easily in the subsequent design phases.

2.1.3 Office Information Retrieval

The problem of retrieving information has to be modeled in a particular way in OIS because information retrieval is an important activity in the office [19–22].

In data base systems, queries are specified completely using artificial language; answering queries uses simple deductive inference from the data. Exact answers to queries are required, dismissals are not accepted in answers, nor false hits; insertions and deletions are usually performed directly by the system.

In information retrieval systems, inductive inference is common; queries usually contain only an incomplete specification of what the user is looking for, and natural language is used for queries; insertion of information is performed by a system administrator, whereas deletions are rarely executed, and approximate answers are considered to be acceptable; therefore, false hits are accepted.

An analysis illustrated by Barbic and Illuzzi [19] shows that neither of the two above-mentioned classes of retrieval models (data bases and information retrieval) is directly adequate for an office environment, but each of them is capable of providing some of the operational characteristics needed for an office system in retrieving stored documents, messages, and data.

Requirements of office retrieval models are illustrated in Ref. *22*. Information retrieval in offices is performed in complex environments and should not be offered simply as an isolated facility. The textual corpus of an office is smaller than that in library applications, but it is unpredictable, highly dynamic, and nonhomogeneous. The users of the office data are also those who make operations on them; therefore, a system administrator may not be compatible with normal office activity because of delays introduced by indirect operation on the system. Moreover, other types of information have to be modeled, in addition to pure textual information.

An issue studied by Croft and Thompson [20] is that of flexibility. It is advocated that the retrieval model should be able to choose the right strategy for a particular user and query, learning appropriate responses with user feedback. The problem is that of considering complex environments and making correct decisions; in general, false hits may be allowed in the office environment, whereas it is necessary to collect all relevant documents.

Another approach to enhance the capabilities of retrieval models is that of using new technologies. The main problem in modeling office data is the quantity of information to be stored, particularly when documents are more than just textual data and also contain images stored as a bit map. The architecture of a system using optical disk technology for archiving documents is illustrated in Ref. *23*. Optical disks offer write-once possibility. Although this is considered to be a disadvantage in conventional IS, it may be desirable

in the office environment because it ensures that information cannot be modified. The system offers the possibility of handling and retrieving documents in several ways (looking at arriving documents, archiving them, and creating new documents, with the possibility of keeping different versions of documents or deleting them). An interesting feature for looking at documents is that of *browsing* through them; this is available only to users authorized to read the documents. Browsing constitutes an appropriate way of retrieving information in office systems, because users may not be able to identify precisely the documents they want.

Appropriate models and techniques are still being studied to solve the above-mentioned problems. It must be emphasized that models for information retrieval are highly dependent on technology. However, some common issues in these models are document structuring, definition of ways of retrieving information from unstructured documents, and types of retrieval operations.

Document structuring is usually performed by defining a few document types, with fixed characteristics, and allowing the definition of subtypes and combination of documents. In many systems, a hierarchical/tree (graph) structure constitutes the basis for retrieval operations. Higher-level nodes of the tree contain more general information (e.g., headlines), whereas lower-level nodes contain the whole document. Unstructured document elements are described in general (e.g., a figure caption), and the content is then represented appropriately. Techniques have been studied for retrieval of unstructured (parts of) documents, (e.g., Ref. *23*).

Retrieval operations are heavily dependent on the model used for the document. If a simple structure, composed of document attributes and a textual part, is used, the attributes may be searched for with data-base-like queries, and the textual part, with information retrieval techniques. If a complex structure is used, the structure itself may be used for retrieval. In the hierarchical structure mentioned above, higher levels may be used to browse through document contents and to filter documents manually or automatically before inspecting their whole content.

2.2 Office Data

2.2.1 Office Document Architecture

The CCITT (International Telegraph and Telephone Consultative Committee) and the ECMA (European Computer Manufacturers Association) are working on standards for document representation to permit the exchange of documents between open systems. The proposed standards Office Document Architecture (ODA) and Office Document Interchange Format (ODIF) allow the representation of logical and layout characteristics of documents [*24*].

ODA provides a way to structure documents, which can be interchanged between different systems both in image form, that is, the documents are received and displayed in the form intended by the originator, and in processible form, where documents are represented in a way that permits editing and layout revision.

Document text is a two-dimensional structure, composed of characters, geometrical elements, photographic elements, and combinations of these. Text and additional control information constitute the content of a document.

Documents have a logical structure, which is a hierarchy of logical objects, such as summaries, titles, sections, paragraphs, figures, and tables, and a layout structure, a tree of layout objects in the form of pages, columns, and areas with contents of different categories. Documents are therefore represented in trees of logical objects, for editing, and trees of layout objects, for presentation on a representation medium. The ODA processing model comprises three steps:

1. Editing: The logical structure of the document is prepared with its content and control information and used, for instance, to specify fonts, sections, and so on.
2. Layout process: The layout process is equivalent to formatting for character documents; the image blocks are combined in an appropriate way. This process is independent of the media used to present the document in its final form.
3. Presentation process: The document prepared in the layout process is presented on a presentation medium, according to the characteristics of the medium.

The formats studied for the ODA processing model are compatible with Teletex and Telex International Organization of Standardization (ISO) subrepertories.

The logical structures that can be defined are complex ones; it is possible to define relationships between logical objects, for instance, cross references to figures or footnotes, relationships between layout objects, such as the overlay order of intersecting blocks and, finally logical layout relationships (layout directives) to define column heads, foot areas, and so on, as attributes of logical objects.

It is also possible to use the ODA representation to define document classes, such as a business letter, a report, a memo, or an order form, which are not standardized but may be defined using ODA documents definitions.

ODA is being used as a basis for ESPRIT Project 121, which studies an ODA/ODIF editor [*25*].

2.2.2 Distributed Office Systems

An aspect that must be modeled accurately is distribution of office data in the organization. In many systems, data are partially stored on a shared database and partially at users' sites on personal workstations. Data are used locally and then transferred to other users or to the central database. Many techniques of distributed databases may be used in office systems [*26*], and others are being studied specifically for the office environment [*27*].

Important problems are concurrent access to shared data, locating navigating objects, and naming objects in the system.

Models may consider distribution explicitly, or distribution may be transparent to the user. When it is transparent, distribution issues are considered mainly at design time, and are not demonstrated in the model presented to users. When distribution is presented explicitly to the user, primitives for accessing remote objects must be provided.

2.2.3 Temporal Information

The ability of dealing with time specifications is an important and characterizing requirement in office systems [*28*].

Time is present in documents, as a date on the documents, as a date or an hour of production of the document, as a period of time in which the document information is considered to be valid, and so on. Time is also necessary to

specify the initiation and completion of activities, procedures, and tasks in the office. Durations of activities, periods of time in which the activities may be performed, time *relationships* between activities, such as the requirement that an activity must be performed before, after, or during another activity, must also be expressed to coordinate work in the office and to meet deadlines.

In office systems, the problem of time representation has been considered mainly to insert time-based alerters to signal particular conditions in an office system [*29*] and to express triggers for activating system activities [*30*].

Temporal aspects in the office environment are considered in Temporal Semantic Office System (TSOS) [*31*]. The basic temporal data types of time extension, time point, time interval, periodic time, and general time are defined as primitive types in the model. A set of operations is defined to combine time specifications. Time types are used in the definition of office elements in the SOS model (see Section 3.1.4) and for time reasoning in an office system [*32*] to check consistency of temporal constraints and to examine precedence relationships between activities, where these relationships are not only specified directly in the system, but are also derived from concatenations of temporal relationships.

2.3 Office Tasks

The starting point for office automation historically has been in the field of data processing, word processing, and document preparation. However, the real opportunities of the available technologies are in integrated systems that automate the office as a whole, rather than providing *ad hoc* solutions for each specific task.

A number of analyses have been performed on office work, aiming to identify the activities that could benefit from the adoption of office system technologies. The result of the study of Harkness [*33*] on the distribution of costs of office work in the United States is illustrated in Figure 1.

The natural observation is that OIS should override the area of supporting document preparation, as only a very limited percentage of total costs is devoted to this specific activity. Instead, as shown in the study of Wilson and Pritchard [*34*], a more articulated taxonomy of office tasks is necessary to capture the actual composition of office work for the various organizational positions (see Fig. 2).

A different taxonomy of office activities has been proposed by Conrath *et al.* [*35*], emphasizing the reasons why they are executed. A classification of office activities using this approach is shown in Figure 3.

An example of the distribution of the activities in an insurance company according to the proposed taxonomy is given in Figure 4.

The approach of Conrath *et al.* is interesting because it introduces a novel element in office task description, the objective of each activity [*35*]. In other words, the attention is given to "what" is done (e.g., filing, sending, etc.) and to "why" a specific activity is performed (e.g., controlling, deciding, evaluating).

Another classification of office activities has been proposed by Strassman [*36*], showing the percentage of time devoted to information handling activities by office workers (see Fig. 5). Again, the portion devoted to manipulation of structured data (i.e., forms), which is easily supported with conventional data processing approaches, represents only a small share of the total.

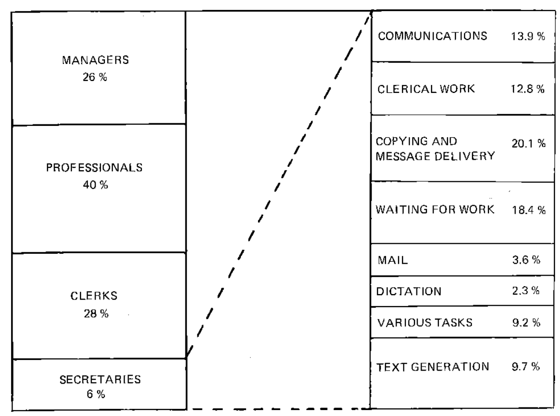


FIGURE 1. Costs of office work (USA, 1974).

2.4 The Intelligent Office

Knowledge-based approaches have been proposed to support office work [*37*], where research results in knowledge representation, problem solving, and decision support are used in office systems. Some examples of models based on artificial intelligence (AI) techniques are illustrated in Section 3.1.2 on process-based models.

The main goal of all these approaches is to have "active" offices, where available knowledge is used to initiate and control office tasks, such as what needs to be done, when, and how, according to office goals defined in the system.

Problem-solving techniques can be used to prepare plans of activities automatically for carrying out office work. Plans can be described at different levels, so complete knowledge can be represented in a concrete way and can be used, when appropriate, to execute office work automatically, whereas partial knowledge can be represented abstractly. Therefore not only can plans be used to start automatic activities but also as guideline and communication means by users.

Another knowledge-based approach views the office as a world of active objects that are able to collect and disseminate information and have associated their own rules of behavior [*38*]. Such objects can be created, can reproduce themselves, and can die in the system when their function is exhausted.

An example of implementation of the concept of active objects moving around in a system is provided by the Imail system [*39*]. Developed at the University of Toronto, Imail is based on the concept of intelligent messages. Mes-

Activity	Top Manager	Manager	Professionals	Clerks	Secretaries
Document Creation					
Writing	10	17	18	7	4
Dictating	6	3	1	—	6
Reading	2	3	3	—	4
Communications					
Reading	9	7	6	3	2
Phone	14	12	11	9	11
Meetings	21	12	7	2	—
Interaction with secretary	3	2	1	—	4
Analysis					
Computing	2	6	10	10	—
Planning	5	5	3	1	3
Use of terminal	—	1	10	6	1
Non-Productive Work					
Mail	6	5	3	—	9
Copying	—	1	1	4	6
Filing	1	2	3	6	5
Searching in files	2	4	4	—	3
Various searches	3	6	6	10	—
Traveling	13	7	2	—	—
Clerical work					
Form filling	—	—	—	8	—
Typing	—	—	—	8	37
Classifying	—	—	—	5	2
Controlling	—	—	—	10	—
Other activities	3	7	11	11	3

FIGURE 2. Percentages of time spent in several office activities.

Activity Code	Description
A	Counsulting, assisting, problem solving, giving instructions
B	Computing, creating invoices, booking
D	Deciding, authorizing
E	Evaluating, verifying, controlling
F	Form handling
G	General administration (manager level)
H	Motivating personnel, allocating human resources
I	Informing, report generation
M	Meetings
O	Orders, requests
P	Planning, budgeting
Q	Appointment scheduling and planning (secretary level)
S	Selling, convincing people, advertising
T	Typing, copying, writing
Z	Activity not classified

FIGURE 3. A taxonomy of office activities.

Activity code	Frequency		Days/year	
	n	%	n	%
A				
A	15	8.3	444	8.42
B	8	4.4	475	9.01
D	38	21.1	1727	32.77
E	35	19.4	763	14.48
F	3	1.7	160	3.04
G	7	3.9	106	2.01
H	29	16.1	681	12.92
I	14	7.8	224	4.25
M	8	4.4	149	2.83
P	16	8.9	402	7.63
Q	3	1.7	23	0.44
S	2	1.1	65	1.23
T	1	0.6	34	0.65
Z	1	0.6	17	0.32
Total	180		5.270	

FIGURE 4. Activity distribution in an insurance company.

	Manager n	%	Professional n	%	Clerks n	%
Structured Paper	106	(6)	282	(16)	352	(20)
Unstructured Paper	598	(34)	1004	(57)	1145	(65)
Oral Communication	774	(44)	383	(22)	226	(13)
Other	282	(16)	91	(5)	37	(2)
Total	1,760		1,760		1,760	

FIGURE 5. Time devoted to information handling activities.

sages are not only used to deliver information but also to collect it and may dynamically route themselves to additional stations, depending on responses that they receive.

The system may be used to build mailing lists of workers interested in particular topics and to deliver messages to other systems and networks of which the user is unaware. They can be used to collect and summarize data on questionnaires, so they are not pure text messages but can be considered programs. The creator of an intelligent message (imessage) writes a program in the specification language used. Rules can be incorporated in the message that will govern the interaction of the imessage itself with its recipients; an imessage can collect responses generated by the recipients, store them appropriately, and eventually make them available to its sender. A statement is attached to the imessage, which describes its recipients, its path history, and so on. The sender can specify termination conditions to determine automatically when a message has completed its tour and prepare the results in the form chosen by its sender.

An example of the use of Imail has been prepared for a Delphi experiment. This is an iterative survey of experts to obtain a consensus answer. The results of the survey are tabulated and sent back to the experts, who can revise their answers according to the other experts' answers. This process is iterated until some criterion is satisfied, for instance, the range or variance of replies. An example of an imessage for a Delphi experiment about the inflation rate is shown in Figure 6.

3.0 OFFICE MODELS

3.1 Conceptual Models

Office conceptual models can be classified into the following main categories,

- Data-based models.
- Process-based models.
- Object-oriented models.
- Mixed models.

Features held in common by each class of conceptual model will be discussed in this section, and some of the most significant models will be illustrated more extensively to provide clarifying examples.


```
        subject A Delphi survey of inflation rates
        set number ?n = 0
        set number ?sum = 0
        set number ?sqsum = 0
        set number ?maxvar = 0.1
        set number ?itresps = 10
        set number ?avg = 8.0
>
What do you think the inflation rate for next year will be?
The last average prediction was ?avg.
        get 1 number
        set ?sum = ?sum + #1
        set ?sqsum = ?sqsum + #1 * #1
        set ?n = ?n + 1
        ?n == ?itresps
            set ?avg = ?sum / ?n
            set !var = ?sqsum / ?n - ?avg * ?avg
            set ?n = 0
            set ?sum = 0
            reship
        ?var < ?maxvar
            terminate
```

FIGURE 6. An example of imessage [39].

3.1.1 *Data-Based Model*

Generally, data-based models group data in forms, which are similar to paper forms in the traditional office. Types of data and operations on data (storage, retrieval, manipulation, transmission) are the basic elements of the office conceptual models. Office activities are then seen as a series of operations on data. Therefore, the main purpose of data-based models is that of representing the office from the viewpoint of objects manipulated by office workers (agents), in a way similar to traditional offices, where work is widely based on documents.

Early office conceptual models were mainly data-based models and were used to design office workstations, where the work of a single user at a time is supported, connecting users through a communication network among the different workstations (see, e.g., OFFICETALK-ZERO [40] and the first version of OMEGA [41]). The general flow of work is not under system control, and single users are supported by the system only in their individual activities.

3.1.1.1 *Office by Example* An important example of a data-based model is Office By Example (OBE), developed at the IBM T.J. Watson Research Center, Yorktown Heights, New York [5, 30]. OBE is a language for describing and manipulating office objects of different kinds; it is an extension of a well-known query language for relational data bases (Query By Example) and of the System for Business Automation [42].

The data structures on which OBE is based are two-dimensional objects, similar to tables. These tables can be relations, forms, reports, hierarchical structures, documents, menus, and so on. In a prototype OBE system, several basic office functions are defined using the same language. Such functions include word processing, querying, automatic triggering of

office activities on temporal conditions and events, data processing, authorizations, communication, and creation and manipulation of documents.

The system allows the user to define (program) its own objects and to use complex data types such as time, graphics, and text. The system has evolved from the consideration of simple forms to the above-mentioned types of two-dimensional structures, although the aspect of objects is similar to that of paper forms.

3.1.1.2 Office Data Model (ODM) An advanced form of an office model based on data is ODM, developed in ESPRIT Project 59, "Minstrel" [*43*]. This model derives ideas from work in the literature on semantic data models and functional models in particular [*44*].

The model uses objects or entities to model "things" (real-world objects) in the office. Such things include office objects such as workstations, filing cabinets, folders, and documents, as well as other objects in the enterprise such as departments, employees, and customers. Each entity may be assigned a unique handle that can be used to refer to that entity.

The model uses the concept of type to organize entities. An entity type may be taken to be a form of specification of a class of objects containing properties and characteristics of the class. An example of declaration of an entity type would be as follows:

```
DECLARE TYPE Person : ENTITY
BEGIN
    DECLARE Name : SINGLE String [1:1] ;
    DECLARE Address : SINGLE String [1:1] ;
    DECLARE TelephoneNos : SET String [0:] ;
    DECLARE Spouse : SINGLE Person [0:1] ;
    KEY Name, Address ;
```

This declaration declares a "Person" type with the properties Name, Address, and TelephoneNos, whereas Spouse is a relationship that is defined from the "person" type to itself. It is possible to declare subtypes of types declared previously.

Operations are defined in the language to manipulate instances of types. It is possible to get the value of the attributes for an instance of a type, to examine all instances of a type, and to update attribute values and relationships between instances.

It may be noted that this type of language has many characteristics in common with functional languages defined for data base schemata. The method of handling instances is procedural and has characteristics similar to data base query languages. Therefore, this model is classified among data-based models.

3.1.2 Process-Based Models

Two types of process-based models are classified in this section: network models and goal-based models. Network models describe sequences of activities to be performed in the office, whereas goal-based models describe mainly relationships between activities and the necessary sequences of activities prepared by the system.

3.1.2.1 Network Model Early procedure-based models were based on a description of procedures carried on in the office. This description could be more or less formal, and it was often based on a graphic representation.

The graphic representation of office procedures was a good point for using these models in the requirements collection and analysis phases during office systems design.

Each procedure is decomposed in steps and is interconnected to one another. Several of these models are based on a networklike representation. The steps of the procedures are linked to one another through triggering conditions that express transitions in performing one step to another.

The first process-based office model was SCOOP [*4*]. Zisman used augmented Petri nets to represent office activities. In Figure 7 a representation of the activities of an editor of a journal is given.

In network-based models, the correctness of the procedure is guaranteed, ensuring that the right steps are followed each time. Exceptions are included in the network when needed.

Information control nets (ICNs) have been developed at Xerox PARC [*45*], to model office activity in a procedural way and to use this modeling to simulate office work and improve the information flow with restructuring operations. ICNs have also been used for flow analysis, deadlock detection, data synchronization, and communication bottleneck detection. An important aspect of an ICN is that the model has a mathematical nature and, therefore, rigor.

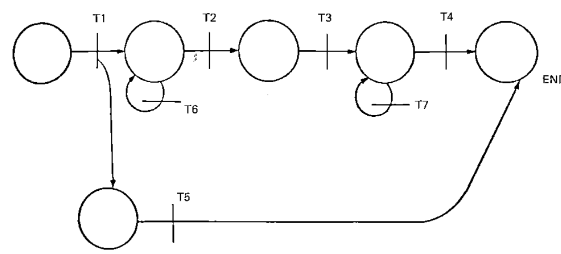


FIGURE 7. SCOOP example.

(T1) Paper received → Send acknowledgement to author and request names of referees from editor.
(T2) Editor supplies names of referees → Instantiate the referee process for each referee.
(T3) All referee activities terminate → Request decision from editor.
(T4) Editor supplies decision on the paper → Generate appropriate final documentation.
(T5) Author withdraws the paper → Termination procedure.
(T6) Editor does not respond within 2 weeks after enabling T6 → Send reminder letter.
(T7) Editor does not respond within 2 weeks after enabling T7 → Send reminder letter.

ICNs are networks for describing information flows in the office. They model procedures such as form completion, filing, copying of documents, and transmission of information, both written and verbal.

Office procedures consist of a set of activities connected by arcs indicating a temporal order, called precedence constraints. Resources in the office include files, calculators, people, pencils, telephones, and so on. In ICN only information repositories are modeled; they include files, folders, forms, scratch paper, people. Circles are used to denote activities, and squares, to denote repositories (Fig. 8). In Figure 8 an ICN describes a procedure for responding to a customer's order letter. The solid circle denotes parallelism, whereas it is possible to specify alternative actions with small open circles. Continuous lines are used for precedence arcs, whereas dashed lines are used for data flows. The possibility of parallel actions and the division among data flows and control flows are interesting aspects in ICN.

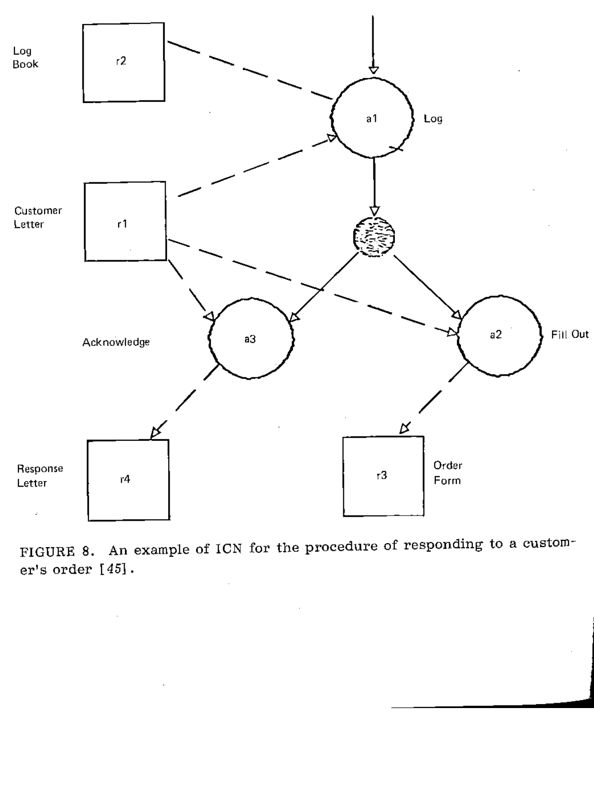


FIGURE 8. An example of ICN for the procedure of responding to a customer's order [45].

ICNs have been defined as a formal model for representing office procedures and data using directed graph of activities and connections with repositories of information. Given an ICN representation of an office system, it is possible to transform, enrich, and correct it by using transformation techniques; in particular, a method called "streamlining" is presented for this purpose [46]. The transformation applies to an initial ICN, developed as the result of the analysis phase, and produces a final, reduced ICN that represents efficiently the basic communication and information requirements of an office procedure.

The method consists in applying three operations in sequence:

- Classification of repositories: Repositories are classified and aggregated according to their type, including data bases, temporary repositories, paper files, and so on.
- Elimination of activities: This analysis involves determining whether an activity is a source or a sink of certain data. The former case refers to activities that create or retrieve data for the first time from a permanent repository. The latter case refers to activities that use data for retrieving or generating other data, for decision making or for storing data in a permanent repository. All activities that are not source or sink activities for some data or that store information redundantly can be eliminated during this analysis.
- Merging of activities: Activities may be merged provided that they remain elementary (i.e., accessing no more than one permanent repository) and respect all the precedence constraints of the original ICN model.

The fundamental information preserved during the streamlining process is the data paths from the source activity of each datum to its correspondent sink activities.

3.1.2.2 Goal-Based Models Another approach to procedure specification is stating the goals to be achieved by the procedure and letting the system determine the correct sequence of steps to be taken. In this type of model, it is not necessary to consider in advance every single case that may arise in the system while the system is still able to handle it; the reasoning capability of the system enhances its flexibility.

OIS developed following this approach use techniques from the area of AI and expert systems.

Two examples of systems are considered based on these concepts: POISE [47] and OMEGA [41].

The POISE system, developed at the University of Massachusetts at Amherst [47], can be used both to automate routine tasks and to provide assistance in more complex situations, following the pattern of work being performed.

In POISE, tasks are represented both in a procedural and a goal-oriented way; the tasks considered are high-level office tasks performed using generic office tools, such as electronic mail, calendar, text editors, and form management tools. The system uses an office procedure formalism to describe the tasks and their implementation using these tools. POISE is useful in environments where the same task is repeated, rather than offering support in completely new situations.

The support that is provided to the user is that of automating some tasks and keeping an agenda of activities, the status of which can be examined using a natural language interface.

A user can specify that a particular task step is not appropriate in certain situations in POISE; this is hard to achieve in systems based on triggering conditions.

Tasks in POISE are specified in a hierarchical way. Procedure descriptions specify the steps typical to the task, the tool invocations that correspond to those steps, and their goals. The system acts as an intelligent interface between the user and the tools available in an office system. The architecture of POISE is illustrated in Figure 9.

The semantic database contains the descriptions of the objects used in the procedures and of the available tools.

The user model represents instances of procedures and data elements in the system through the user's state; the user's state contains parameters with values that account for instantiations of procedures described in the procedures library; these instantiations are derived from users' actions. The user model also contains the instantiations of semantic data base objects.

A procedure in the procedure library can be specified like the example presented in Figure 10. The procedure description contains a (high-level) algorithm for the procedure, a list of its parameters, a list of conditions to be met to have a valid instantiation of the algorithm, and preconditions and goals. These goals can be used to start procedure instances automatically.

The POISE system can be used in two different modes of operation: interpretation or planning. In the interpretation mode the user invokes tools directly, and POISE attempts to recognize the user's goals in the context of the procedure library (goals are used in this context to identify completion of procedures). In the planning mode, the user invokes a procedure, and the system must then execute the procedure as far as possible, based on the procedure's goals.

In the interpretation mode, the user is presented at the top level of the POISE system with a menu of tasks (such as filling out a particular form); all accesses to tools must be done through POISE menu choices, thus enabling the system to monitor tool usage. Every command to and response from a general tool is intercepted by POISE, though it appears as if the user is interacting directly with the tool. By intercepting a message, the system can check to which activity in progress it can be related and, if one is found,

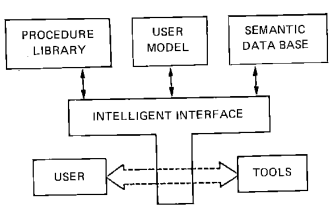


FIGURE 9. POISE architecture [47].


```
PROC        Purchase_Items
DESC        (Procedure for purchasing items with nonstate funds.)
IS          (Receive_purchase_request)
            '(Process_purchase_order | Process_purchase_requisition)
            '(Complete_purchase)
WITH        ((Purchaser = Receive_purchase_request. Form. Purchaser)
            (Items = Receive_purchase_request. Form. Items)
            (Vendor_name = Receive_purchase_request. Form. Vendor_
            name))
COND        (for_values {Purchaser Items Vendor_name}
                (eq Receive_purchase_request. Form
                    Process_purchase_order. Form
                    Process_purchase_requisition. Form
                    Complete_purchase. Form))
PRECONDITIONS-
SATISFACTION        (for_values | Purchaser Items Vendor
                        exist Complete_purchase. Form))
```

FIGURE 10. POISE procedure specification [47].

verify whether it is appropriate (error conditions and exceptional ways of carrying out a task are both considered; appropriate messages are sent to the user performing the task). If more than one interpretation is possible, all interpretations are maintained, but there are some heuristics criteria to determine the most plausible interpretation; if this selection turns out to be incorrect, backtracking is performed and the next most likely interpretation is selected. If one procedure is invoked instead of a similar one (e.g., Full_Purchase_Requisition, instead of Full_Purchase_Order), the user is advised with a possible alternative action, and an explanation of the possible error in selecting the form is given. This feature can also be used to monitor data entry, with possible corrections.

POISE's natural language help facility may be invoked to make inquiries about the state of active, completed, or expected activities.

The planner enables tasks to be carried out or completed automatically by the system; it can also be used to start tasks up to the point in which the intervention of a worker is needed.

A goal-oriented representation can be useful in supporting decision making. The deduction mechanism may not only be used in controlling task sequences as in POISE but also in reasoning about change and contradiction, as in the system OMEGA [41].

In OMEGA, the office workers' activities are supported as the system provides a way of analyzing the reasons for contradiction when they are reached, using a deduction mechanism. Viewpoints are employed to limit the effect of contradictions, and in the reasoning process the system explicitly keeps track of what it believes to be true (assertions) and why something is believed to be true (justifications). An example of such a system is provided by Barber [41], where the system is controlling the possibility of a certain employee being assigned to a certain job. The global goal of establishing whether the proposal is reasonable is decomposed into several subgoals. OMEGA was first used as a form management system, based on AI concepts; it was then employed to model more complex office activities, involving decision making in the office.

The model consists of stating a series of facts, defining deduction rules from facts, and allowing the attachment of assumptions to facts. All these elements are then used by the reasoning mechanism.

Models to specify office procedures using AI techniques have the disadvantage of being inefficient and also of hiding the flow of activities in the organization. Rigid procedures, in particular, a mixed approach, where some of the procedures are specified procedurally and others are specified through their goals, would provide a more effective representation. This is the type of approach already taken in part in the POISE system, where goals and procedural parts of activities are specified separately as needed. This causes a rigid separation of use of these two types of procedural specifications: A more integrated way of doing this would probably be more effective for an office system.

The Office Advisor is being developed within ESPRIT Project 82. It is based on the use of a knowledge representation language (KRS), which allows the specification of concepts in an inheritance hierarchy. Office Advisor distinguishes three levels in this hierarchy: the formalism level, where all basic types of knowledge, such as rules, relations, functions, frames, values, and so on, are defined; in the next level, the office level, types in the office world are defined; in the third level, instances of types are represented in the application-specific or concrete level.

Office work is considered a goal-oriented activity and is supported by planning and problem-solving activities. For each goal, inputs, preconditions, postconditions, subgoals, and so on, are defined. When a goal is activated in a concrete context (for given data), it is expanded to a planning tree, which represents a hierarchical plan for the task, where the top of the hierarchy describes the main goal and each successive level describes increasingly detailed subgoals. A solution is constructed by piecing together results of subgoals. Each goal has an associated monitor, which is in charge of building the list of appropriate subgoals for the current context and of determining the strategy to be used in the exploration of subgoals and in combining the results.

The Office Advisor subgoals can start activities in the system, collect data from the user, and issue reminder messages. A goal activated for a certain situation may be valid over a long period of time, before it is completed through its substeps.

In the example given for organizing trips in the company, subgoals involve verification of appropriate funding and paperwork to obtain all necessary trip authorizations and to prepare statements of expenses in due time.

3.1.3 Object-Oriented Models

A concept that is popular now is that of "office objects". The model of objects is similar to the one of object-oriented programming languages such as SMALLTALK [48]. Object instances of one or more types or classes are defined. The behavior of its instances is defined for each type by specifying a set of operations that can be applied.

OPAL [49] is an application development environment for office information systems. It is based on an object management approach, and the central concept is that of packets, which contain data and actions.

In this system the inheritance mechanism among types is studied in detail. Property inheritance is typical of this type of model but sometimes can be ambiguous, as is shown in the following example (Fig. 11):


```
PACKET a;
    ATTRIBUTE x,y:INTEGER;
    BODY s1;s2;
    END;

PACKET b;
    ATTRIBUTE x,z:INTEGER;
    BODY s3;s4;
    END;

a,b PACKET c;
    ATTRIBUTE y:INTEGER;
    BODY s5;s6
    END;

p:PACKETREF(c);
p <- NEW(c);
```

FIGURE 11. OPAL inheritance mechanism [49].

PACKET *c* inherits properties of *a* and *b*; the problem is how to put bodies in sequence and how to deal with name conflicts (e.g., attribute *x* in Packet *a* and *b*, or attribute *y* in PACKET *a* and *c*). Part of the model is to set up rules for the resolution of this type of conflict.

3.1.4 Mixed Models

Mixed models are explicitly assuming more than one type of element as the basis for system specification and are defining the relationships among these elements.

An example of a mixed model is SOS (Semantic Office System), a model developed at the Politecnico di Milano [50]; the SOS model for the formal description of an OIS uses the SOS formal description language. SOS classifies office elements into three different submodels: the static submodel, the dynamic submodel, and the control and evolution submodel.

- Static submodel: Static aspects, such as all office information contained in documents, can be described considering structured and unstructured parts, specifying different document types; relationships between documents can also be described. All system information, such as events occurring in the system, are also stored in documents (system documents as opposed to user documents). Besides document description, the static submodel includes the description of office agents, which are handling documents and performing activities described in the dynamic part.
- Dynamic submodel: Dynamic aspects include all office activities (automatic and manual); these are described in different levels of detail and can be composed to form more general activities. A few primitives for document manipulation are defined as a basis for activity definition.
- Control and evolution submodel: Control aspects include the possibility of specifying, using control rules in the form "conditions in documents—activities," all types of control on the sequence of ac-

tivities and operations on documents. Activities are started, finished, and synchronized through control rules, which also control document access and activity performance rights. Abnormal situations are detected through rules and checking document content and activities state.

Control aspects include a control over the evolution of the structure of the office; in the model it is possible to define new types of documents, agents, and activities at operation time, but this creation of new types has to be controlled to maintain consistency with already existing types.

A characteristic common to all SOS submodels is the use of the concept of abstraction. The generalization (inheritance from super-types), association (grouping of elements of the same type), and aggregation (grouping of elements of different types) abstractions are used, both in the SOS specification language and in the language for manipulation of instances.

The result of the SOS method is a description in a SOS model of all the above-mentioned types in the OIS being designed. The purpose is to specify office element semantics with their structure.

Most of the recent office models belong to the mixed category, because this type of model allows more complete specification of different types of fundamental elements in the office and of their interrelationships.

It should be noted that in the early days, some of the models belonging to one of the other categories mentioned above have recently evolved to the mixed category. An example of this fact is OFFICETALK-D [45], which has merged a data-based model like OFFICETALK—ZERO and a process-based model like ICN into a single system for simulation and control of office work.

3.2 Logical Office Models

Most of the models mentioned above are mainly either specification models for conceptual office description or prototype systems using this type of model in their activity.

An interesting evolution are models that represent different office tools and give the possibility of combining different building blocks in the same system. For instance, it is possible to define some common characteristics to all the text editors, thus modeling them in the same framework, and then plugging them into the system being built. We will call this type of model logical office models, equivalent to logical models used in data base design or in IS design.

During the design of an office system, it is most important to obtain a rapid response from the user to check whether the system satisfies his/her requirements, and it is also important to evaluate the technical solutions to be adopted to build the operational system. Logical models can be made executable to test parts of the OIS being built.

By modeling each component, it is possible to provide two types of design aids to be used in parallel in the design of a complete office system. A user model can be employed to simulate the functioning of a particular system module as for user interaction; this module can be introduced in a prototype of the system being built for user evaluation. An architectural model can then be employed to assess how building blocks can communicate (and their compatibility) and to simulate performances of the system (response times, identification of bottlenecks).

Although some models have already been developed for software tools (although not specifically addressed to the framework of computer-aided design of an office system, they could still be used for this purpose), architecture models to be used for office modules are still to be defined.

3.2.1 *Tool Interaction Models*

Some systems have been developed to enhance the performance of a typical office tool to support office work. Examples of such systems are ENSAMBLE, to support cooperative decision making in a distributed office system [*51*]; Imail, to provide automatic features to messages in an electronic mail system [*39*]; form management systems, to model office activity based on the concept of forms, which is also the method of communicating among office workers [*9*]; and MACROS, which supports automatic production of software to interact with an office system based on a formlike interface [*52*].

A system that provides access to different tools is POISE, already described in Section 3.1.2. This system supports high-level office tasks with generic office tools; the implementation of office tasks is done using the available tools. In POISE, the user/poise/tool interface consists of a description at a logical level of each tool and how it interacts with the user and with POISE. Tools are grouped into classes according to the style of interaction with the user: menu-based, form-based, command-line, graphic, and natural language. Other tool information concerns the way data are passed to the tool from POISE or from the user, and vice versa, and the corresponding actions. In the language proposed by Croft and Lesser [*53*], primitives are defined for each tool class; for instance, for a form, field names and corresponding POISE action names are defined, basic form handling procedures are specified (i.e., enter field, scrolling up/down the form, movements from one field to another, movements to the previous/next form page, movements to the top/end of the form), and field positions are specified. Similar specifications are provided for menu and command tool classes. The graphic and natural language tool classes are still under study in the system.

3.2.2 *Architecture Composition Models*

Specific architecture composition models for office systems have not yet been proposed. Systems such as ICN [*45*] allow the simulation of office activities (considering frequencies of activities and activity sequences) to monitor office work and to determine bottlenecks. In this simulation, some parameters are fixed for activities duration, because it depends on architectural choices; these choices, however, are not shown in the model.

Many studies have been conducted on information sharing in the distributed data base area; data models and design choices have been studied widely. Issues such as where to distribute data, how to interact with global data at other sites, and how to coordinate concurrent usage of data, which are already studied in the distributed data base field, could also be applied to office systems in which information is shared and the system is inherently distributed. However, additional considerations are necessary in order to extend these studies to office design. Among these, different types of data should be handled in the office (mainly documents, including unformatted information and images), and a different emphasis should be given to communication among users (communication is usually hidden as much as possible in distributed data bases).

Several simulation models have been proposed in the area of computer networks [54]; these models consider different network configurations and simulate the network under possible workloads. Interaction among office modules could be simulated in a similar way to make appropriate design choices based on systems performances.

Recently, methods of interconnecting personal computers and evaluating them have been proposed [27]. Work based on personal computers has many similarities with that performed in an office system. Each user has some private information used in local work, shares some global information with other users, and transmits some personal information. The link via a communication network is weaker than when using distributed data bases, so that explicit mailing facilities are used to communicate among users. Usually, however, when considering a network of personal computers, each user is assumed to be working autonomously. On the other hand, office systems need to consider office work as a set of procedures carried on by several workers, whose activities are coordinated.

Hence, new specific research is still needed in architectural modeling of office systems.

4.0 OFFICE DESIGN METHODOLOGIES

Although many conceptual models have been presented to describe OIS, less attention has been paid to the development of complete design methodologies specifically conceived for office systems. Three sample OIS design methodologies are briefly illustrated here: OFFIS, Office Analysis Methodology and MOBILE-Burotique.

Because the OFFIS methodology is based on a tool for requirements specification analysis, it is presented in detail in Section 6.0 on environments for office systems specifications.

4.1 Office Analysis Methodology

Office Analysis Methodology (OAM) has been developed by the Office Automation group at MIT [3]. OAM is based on the analysis of activities performed in the organization. The goal of OAM is to understand office work in terms of functions, activities, flows, tasks, and so on, and to specify this knowledge in a formal way. Therefore, OAM provides the analyzer with a conceptual model of office work to be used in the subsequent phases of optimization of office procedures and of implementation. OAM is directed at the analysis of semistructured problems at a managerial level in order to understand the business goals of the organization.

The results of the requirements analysis phase conducted following OAM are specified in the OSL language for office description. This is a high-level and problem-oriented language that has embedded the concepts of hierarchical abstraction and resource. The advantage of using OSL is that of having a formal and nonambiguous description of the office.

OSL is based on the concept of office function. Each function contains the formal specification of initiation events, a body, and termination events. The resources needed for performing the function are specified, that is, people, equipment, and documents.

OAM is a methodology for specifying functions. Requirements are collected top-down, refining functions at different iterations, whereas the process of integrating the office system with the organization and other systems is performed bottom-up.

OAM provides mainly guidelines to collect a user's requirements. It stresses the importance of the user's participation and acceptance of the system before its implementation.

The process of office analysis involves the participation of all the managerial staff, from office managers to some of the personnel (Fig. 12). The purpose is to understand why functions are performed, what they do, and how they are implemented. Each function is divided into three tasks: initiation, management, and termination. The first step of the methodology consists of identifying and listing the resources used and the three tasks for each function. The second step analyzes the procedures performed in the office without considering exceptions. The third step classifies exceptions in main categories. The fourth step identifies conflicting situations, exception handling, and *ad hoc* decision-making processes. The fifth step consists of a revision of the office model built in the previous steps, involving validation of the model by office managers, definition of user interfaces, and general exception handling performed in office work. The last step consists of the preparation of the documentation.

OAM looks at the office with an organizational model. Office functions are examined top-down, first interviewing the office manager and planner and then examining office activities in greater detail following the office hierarchical level. The integration of the office system with the organization and the other systems is performed bottom-up.

OAM provides a set of guidelines on how to perform interviews to collect the requirements, rather than tools, for OIS implementation.

4.2 MOBILE—Burotique

The "Organization—Methodology" group, part of KAYAK project at INRIA (France), has established and experimented with the MOBILE—Burotique methodology for the design of OIS [55]. Human and organizational factors are considered together with the technical part of the design. The characteristics examined in the office, to which the system being designed must refer, are mainly the evaluation of the productivity of the office, economic evaluation of the proposed technical solutions, and acceptance of the system by the organization and the office workers. These factors considered in the office imply a series of consequences in the technological part of the project: an OIS must be modular, that is, composed of integrated functional modules, it must be simple to use, and must provide a series of measures of the work performed.

MOBILE—Burotique is a metamethodology, that is, it provides a series of types of instruments for observation and analysis. The instruments can be classified into instruments to do a preliminary observation of the organization, instruments to collect information about the procedures performed in the organization and, finally, instruments for supporting office analysis, like simulation.

Some constraints are put on the implementation of these instruments: their cost, the usefulness and reliability of the obtained results, the time required to use them, and how they are accepted by the organization. They are then chosen with a cost—benefit analysis.

This methodology, more than the other two, takes into account social factors in system design. Moreover, it provides some guidance in each of the OIS design phases. The methodology is composed of six steps:

1. Intelligence: Objectives and limits of the study in the organization and a selection of a representative set of employees are done.
2. Diagnosis: Data are collected to evaluate the existing situation and to identify problems.
3. Functional choice: A functional solution is proposed, providing a conceptual description of the application, and it is submitted to users for validation.
4. Technological choice: Configuration selection is done, with the help of a cost–benefit analysis.
5. System implementation.
6. Evaluation.

Most Mobile instruments are for data collection. ICN is used as a conceptual model for the description of the functional choice.

1. Meet with the office manager

   Organizational context and reporting relationships
   Functions and resources of the office
   Identification of conceptual objects and procedures
   Identification of key personnel

2. Produce initial procedure descriptions

   Conceptual objects
   Core procedure steps and major alternate control paths
   Inputs and outputs
   Data bases
   Environment and special equipment

3. Develop and analyze a draft description

   Examine for inconsistency and incompleteness
   Construct list of exception possibilities

4. Iterate the interview process

   Circulate draft description
   Resolve conflicts and ambiguities
   Investigate exception-handling procedures
   Watch for *ad hoc* decision making

5. Review the analysis with the manager

   Validate intentions behind each procedure
   Clarify what happens at interfaces with other offices
   General exception handling

6. Finalize the office description

FIGURE 12. Structure of the analysis procedure in OAM [3].

4.3 OSIRIS

OSIRIS (Office Systems Information Requirements Integration and Specification) is a methodology for the specification of procedures in an office information system, developed at Politecnico di Milano [56]. It does not address the important problem of the preliminary analysis of the office environment, aimed at determining the necessity of innovation or reorganization of the office information system. However, once that preliminary analysis has identified the necessity of an in-depth analysis of the office system, a formal and structured method is given for building the specification of such a system. The most relevant aspects of this methodology are the following:

1. OSIRIS uses a model for representing an OIS that incorporates elements selected from the SOS model, also developed at the Politecnico di Milano [50], and from ICNs [45]; the emphasis is not in determining new features but in selecting and integrating them. A graphic representation has been developed for all the elements of the model, which improves its user friendliness.
2. Following the practice of design methodologies in the area of IS and data base systems, the design process is decomposed in two phases: independent design of office procedures and integration of office procedures [28]. Procedures incorporate activities that are strongly interrelated, for instance, because they are under the responsibility of the same manager (similar to "views" in the design of data bases [2]). The rational for this decomposition is that the analysis of the same procedure should be conducted by the same analysis team in a unitary way. The integration of different procedures leads to the development of an integrated office system model and, more important, to a deep understanding of the interrelations and synchronizations between different procedures.
3. The design of a single procedure is accomplished by a progressive deepening of the level of detail of the collected requirements; this process corresponds to the interaction of the analyst with different levels of the office organization (e.g., top management, middle management, and end users). The enrichment of the description is supported by a top-down refinement mechanism, which can be applied to all the elements of the model.
4. The integration of office procedures is based on recognizing their need of synchronization, because one of them uses some information that is produced by another one. This process is done bottom-up, beginning with a detailed description of the office environment.
5. The refinement of the integrated office model leads to the development of a minimal form, called "canonical form," which incorporates all the information on the synchronization between different procedures and, at the same time, uses the minimum number of elements.

4.4 Rapid Prototyping of OIS

Rapid prototyping has been used in software engineering to get rapid response from the users about the correctness and completeness of system requirements. The rapid prototyping of office systems aims at getting the user's feedback in the early system design phases, before system implementation; such a rapid response is useful to avoid wasting time and money in inappropriate

system developments. Rapid prototyping is a technique that appears promising in the development of OIS, as a method for assessing user needs, particularly when nonprocedural tools are used during office work, such as decision support systems, data base management systems, electronic spreadsheet systems, telephone, electronic mail, word processing tools, and office work support systems in general [57–59].

Rapid prototyping has recently been used in software engineering and IS development. Usually, this approach is considered as an alternative to the traditional development life cycle, because its main characteristic is to make a working prototype system available as early as possible. Therefore, the long phase of requirements collection is shortened in order to get the user's feedback before completing the requirements collection phase; the exercise of the prototype by the user then allows collection of further and more detailed requirements.

Prototyping has been subdivided into four steps by Floyd [60]: functional selection, construction, evaluation, and further use.

Functional selection is performed to choose which functions should be included in the prototype. The range of features offered by the prototype is never the same as that offered by the final system; otherwise the protytype would be the system itself. Two types of functional selection can be performed:

- Vertical prototyping: Only some of the functions are selected, but these are developed to their final form.
- Horizontal prototyping: The functions are not implemented in detail, but they work only with a subset of data, or are simulated, and parts are omitted; this type of prototyping is used to demonstrate functions.

Usually, the selected approach is a combination of these two techniques.

In the construction of the prototype, the emphasis is on providing a system that will function as soon as possible. Therefore, aspects such as efficiency, reliability, and data security are usually neglected and are implemented only in the final system, unless they are among the functionalities that have to be tested with the prototype.

Evaluation is critical, because it should provide the required feedback to the designer of the system. Evaluation can be performed by a user or a group of users, and in order to be meaningful, appropriate training has to be provided to the users before they exercise the prototype.

Further use of the system depends on the experiences gained with the prototype and the development environment; either the prototype is only used to gather user feedback and the system is reimplemented completely, or the prototype or parts of it are used as a kernel for the system to be implemented.

4.4.1 *An Approach to Office Prototypes*

Early experimental studies indicate that fifth-generation languages and tools will support the formal specification of office systems effectively. Section 4.5 discusses the TODOS Project. The adopted logical model is the basis for rapid office prototyping in TODOS, and it is thus validated in coordination with the user. A precondition for this time-and money-saving feedback from the user in this early phase of office system development is the capability of modeling a user-oriented representation of the specified office system. Therefore, the

goal of the work in TODOS is the development of a tool for rapid prototyping based on fifth-generation languages.

The requirements are specified in a language that is strictly related to the conceptual description of office elements defined in the logical design phase. From this description, the prototyping tool will be able to derive how the user will interact with the system, building the necessary programs, using artificial intelligence techniques to synthesize programs and using specific languages to provide formlike interfaces to the system.

A set of primitives for developing office systems is identified as a basis for the prototyping language: basic communication primitives, document handling and exchange primitives, authentication primitives, message handling primitives, archiving primitives, data base access, and so on.

The prototyping language will also include the possibility of describing requirements for the user interface to the system to provide a user-friendly interface in the prototype. The development of the final system should be based on the results of the predevelopment phase, and the prototype of the office could become the office system kernel.

4.5 TODOS: A New Environment for OIS Development

The TODOS (Automatic Tools for Designing Office Systems) project under the European Strategic Program for Information Technology (ESPRIT) has the aim of developing an integrated method for designing office systems and implementing computer-based tools for supporting this method in a unique framework [61]. Its goal is that of making OIS development easier, quicker, and more reliable by providing a set of design tools to analysts, designers, and users. Techniques for collecting design data, for rapid prototyping, and for choosing system architectures are studied in the project.

The TODOS office design methodology is subdivided into four phases (Fig. 13):

1. Requirements collection and analysis.
2. Logical design.
3. Rapid prototyping of office systems.
4. Architectural design.

4.5.1 *Requirements Collection and Analysis*

The requirements collection and analysis phase supports a preliminary study that considers the feasibility of the introduction of advanced technologies in the office, based on a general evaluation of costs and benefits.

Several types of information about work in the office, needed resources, and layout of the office must be collected. From the analysis of the requirements, those activities in the office work that are critical or more suitable for applying new technologies are selected as the main object of the subsequent phases of the methodology.

Specific tools for this phase must first allow the collection of data and requirements about the office being studied and then their analysis in order to uncover inconsistencies in the data and to individuate possible problems in the organization.

The tool being studied for requirements collection supports a computer-based office data dictionary.

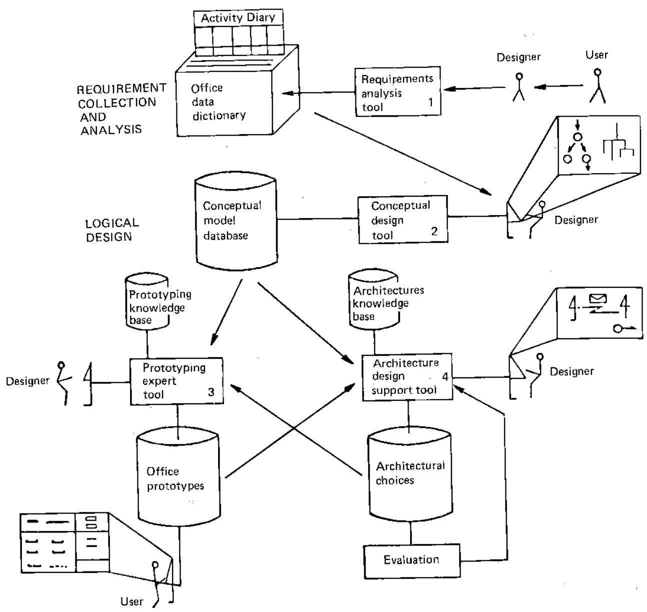


FIGURE 13. TODOS design methodology with automatic tools [61].

4.5.2 Logical Design

The objective of the second phase is to provide a conceptual model for logical design based on the information collected in the previous step of the methodology. The conceptual model is the basis of the subsequent phases of architectural design and office prototype design.

The TODOS Conceptual Model (TCM) has been studied during the first months of the project. In its preparation, an effort has been done to distinguish between entities that correspond to concepts in the office (called objects) and physical entities (documents, messages, agents). The dynamic elements in the office are based on the notion of events that cause, through actions, transformations to objects and entities. Abstraction forms are used in the description of the different elements: classification, generalization, aggregation, and association. TCM derives concepts both from the SOS office model [50] and from the Remora methodology for information systems design [62].

The TODOS Specification Language (TSL) will be the basis for the development environment supporting the methodological steps for logical design of office systems.

In this phase of the methodology the principal characteristic of the tools under study is that of providing an effective user interface to describe the model, with easy-to-use graphic facilities. In addition, tools will be provided for evaluating, separately and together, the document flow and the activities flow and to verify consistency and completeness.

The specifications expressed in TSL will be stored in a specification data base. The specification data base is the principal tool to be used in conceptual office system design. It will be possible to access it with an appropriate query and manipulation language in order to extract reports on the specifications and to investigate the different characteristics on the specifications. The query language will also be used as a basis for a tool to verify consistency of requirements.

A graphic interface to the specification data base will be designed and implemented. This interface will represent document structures, document flows, and activity sequences graphically at different levels of abstraction. Figure 14. illustrates an example of graphic representation of a view on the conceptual model specifications (see Ref. *56*).

Basic functions provided by the graphic interface will be visualization of the existing model, (graphic) insertion of new elements in the model (guaranteeing correctness and completeness), deletion of elements and modification of elements and flows.

Different levels of abstraction will be provided to view the office in general, to investigate some of its parts in more detail (e.g., a department), or to view only elements connected to a certain element (e.g., documents related to an activity, documents flows).

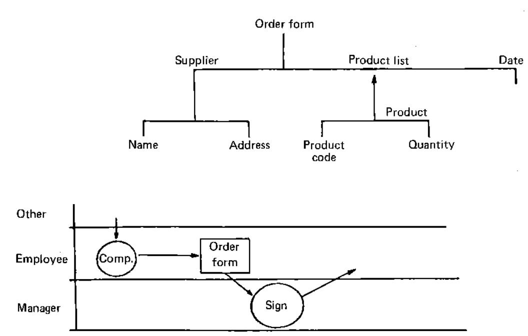


FIGURE 14. Information displayed from the conceptual model.

4.5.3 *Rapid Office System Prototyping*

The goal of the prototyping phase is the development of a rapid office prototyping tool to support OIS development.

In various cases, though the design process is methodologically supported and specific support systems have been developed, the outcome of OIS development does not always satisfy the user's requirements. The late verification of requirements, obtained only after system implementation, may cause the waste of a great amount of time and financial resources due to incorrect requirements interpretation. Therefore, rapid prototyping has been adopted as a technique for allowing the user to verify whether the system meets its requirements during the first development phases.

4.5.4 *Architectural Design*

One of the most difficult problems is the mapping from the user's requirements and functional specifications to a precise hardware and software architecture.

Existing OIS development methodologies in the literature devote little attention to this problem. In other system development areas, several tools have been developed to assess advantages and disadvantages of architectural design choices. Analytical and simulation models have been used, and the parameters to be taken into account have been studied; this has been done particularly in the network design area.

In the architectural modeling phase of TODOS, the components and activities of an OIS identified and defined in the conceptual modeling phase (using the language and tools realized in logical design phase) are mapped into hardware/software components of several alternative architectures, implementing the target OIS. Therefore, languages and tools must be developed to support the system designer in the choice of the most appropriate architecture and in the evaluation of the hardware/software modules of the final system before purchase or implementation.

As a basis for architectural design, existing hardware/software modules are being classified, identifying the most significant elements for each type of module and studying parameters that can describe and qualify the hardware/software modules examined. For example, characteristics and capabilities to be described in the architectural model of a user workstation could be the supported operating system and languages, the screen, the pointing devices, the processor speed, the memory and storage size, the networking interfaces, and so on.

The kinds of components that are needed for OIS include communication facilities, distributed functionalities, user workstations, special-purpose devices, and hosts (servers). The model should be able to describe both existing modules and modules to be implemented for *ad hoc* applications.

The architectural model specification language is being studied and tools are being implemented to support the model. Architectural choices expressed with the architecture specification language are stored in an architecture database.

A simulator is used to evaluate the global system behavior of the proposed architectures. The results of the simulations are supporting the choice proposed architectures. The simulator should give an idea of the performances offered in relation to response times, bottlenecks, physical document flows, and so on.

A graphic interface is provided to the architecture database to present different architectural choices to the designer. The graphic interface is also used to create/modify proposed models and to visualize results from the model simulation (e.g., showing bottlenecks).

5.0 REQUIREMENTS SPECIFICATION METHODOLOGIES

In this section, requirements specification methodologies, applied to software products or information system design, are presented; office methodologies, as discussed in Section 4.0, must consider some new aspects or emphasize aspects that are not so important in information system design. However, all existing (and future) office methodologies will probably have much in common with conventional requirements specification methodologies. SASS, SADT, and JSD will be examined here, as they are cited more frequently in the literature.

5.1 Structured Analysis and Systems Specification

Structured Analysis and Systems Specification (SASS) is a systems analysis methodology described extensively by De Marco [63] for the analysis and formal specification of a software system to be used in the first phases of the software life cycle.

The goal of SASS is to provide a formalized and yet user-friendly specification of the system being developed to facilitate the communication between designer and users in order to obtain users' validation of specified requirements. The specification of requirements is the basis of the contract for the work to be performed, so it is necessary that it is well understood and agreed on by the user and the designer.

SASS uses data flow diagrams, a data dictionary, structured English, decision tables, and decision trees as tools for representing specifications.

Data flow diagrams (DFDs) represent the information flow in the organization. Input data are transformed to output data through a series of transformation processes. Graphically, processes are represented with bubbles, and the data flow, with vectors. An identifier is associated with each element in a DFD. Both vectors and bubbles have a name that must be self explanatory so that diagrams can be understood by users who must validate them.

The given name is particularly important for vectors representing data flows, because it is also reported in the data dictionary, which is a collection of information about all the data flows in the system and provides detailed information about their composition.

In addition to a name, bubbles are also given an identification number. This is important because DFDs are hierarchical: bubbles in a DFD can be expanded into separate DFDs, each identified by the original bubble number, describing the process of information transformation of the bubble with a new DFD in greater detail. Numbers are assigned to bubbles and DFDs with a dot notation, for instance, DFD 2.1, originated by bubble 2.1, includes bubbles 2.1.1, 2.1.2, and so on. In general, a limited number of bubbles is allowed in each DFD (maximum, seven); if more bubbles are needed, some of them could be grouped in a separate DFD. The idea is to have a hierarchical description of the system, beginning with a high-level abstract description of the data transformation in the system, until a detailed description of the information processing activities is given in the lower-level diagrams.

Bubbles that are not further expanded to DFDs are described in a formal way through three different alternative tools: structured English, decision tables, and decision trees. The purpose of these tools is to represent

the processing of the bubble in an unambiguous, but not too formal way, so it can be understood by the user.

Structured English uses the terms specified in the data dictionary plus a restricted set of words, such as "for each," "if," "otherwise," "while," and so on. Although this set of words is not defined *a priori*, it must include all three basic elements of structured programming: sequence, alternative, and iteration.

Decision tables and decision trees are equivalent and are used to represent a complete set of alternatives in a formal way, giving the actions to be performed in each case; these tools are appropriate, particularly when many alternatives are possible, because they provide a way of checking whether all possible cases have been considered.

5.2 Structured Analysis and Design Technique

Structured Analysis and Design Technique (SADT) is the name of SofTech's proprietary methodology based on Structured Analysis (SA) [*64*].

SA is an approach similar to that of SASS, illustrated in the previous section. SA is based on structured decomposition (Fig. 15), and it allows a rigorous decomposition into units carried out to any required degree of depth, breadth, and scope, with rigorous and precise interrelations.

As in SASS, the goal is that of representing user needs and communicating those requirements to those who must produce an effective system solution.

An SA model is a structured collection of diagrams, designed according to precise rules and graphic syntax, integrated with natural language descriptions (40 SA language features are presented in Ref. *64*).

To illustrate SA features, an example of an SA diagram is shown in Figure 16. When modeling activities (in addition to data flows defined in SASS diagrams, represented as [left] input arrows, and [right] output arrows of each block), arrows in the diagram allow the specification of control flow (down arrows), and mechanism to be adopted for the block (up arrows).

More information is added with the automation of diagrams, such as author, title, notes, and indications of suggestions of modifications to diagrams by readers of diagrams. Both authors and readers are assumed to be reading and modifying diagrams; the SADT methodology illustrates how the review process of diagrams should be carried out, having the diagram read and commented on by readers.

Diagrams can represent both activities and data flows with the same notation, as illustrated in Figure 17. Therefore, cross-checking can be performed on diagrams to consider consistency and completeness of the information they contain.

5.3 Jackson's Structured Design

Jackson's System Development (JSD) is a method for specifying and implementing computer systems [*65*]. It covers the phases of requirements specification, functional specification, logical system design, and application system design in the software development life cycle.

JSD is a bottom-up design methodology. It starts with modeling the world: Entities and the actions performed on these entities are first identified from natural language requirments specifications; actions can exist in the system only through entities. The model of the world is then constructed

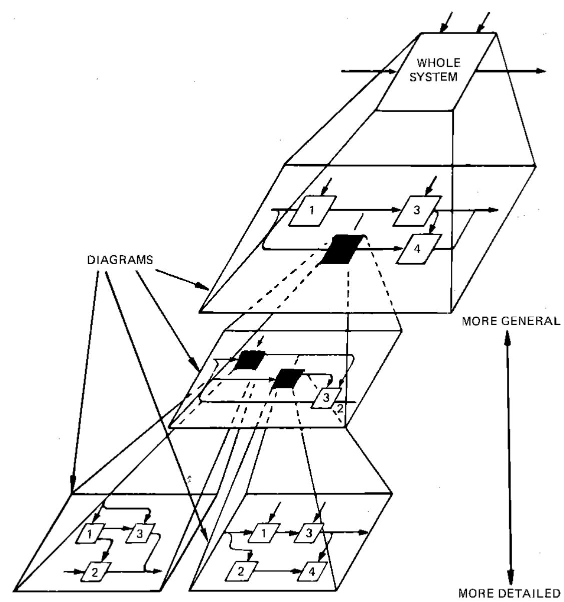


FIGURE 15. Structured decomposition with SADT [*64*].

from the first informal specifications, structuring all actions that can be performed in the system in structure diagrams. A structure diagram is given for each entity which illustrates in a graphic way all the actions that can be performed on the entity, according to their temporal sequence.

The time factor is essential in JSD; it is possible to use the methodology only for systems where the concept of evolution in time is present; whether the order of events is fixed or variable, it is always of central interest and importance (dynamic world). JSD is not tailored to design systems for static realities, such as, a national census information system.

In structure diagrams, actions on entities are specified with their possible ordering. The sequence, iteration, and selection constructs are, as always, present. An example of a structure diagram in Figure 18 illustrates actions that can be performed by a customer entity in a bank.

The initial model of the world is mapped to system specification diagrams (SSDs), which illustrate the modeling of the world on which the system to be

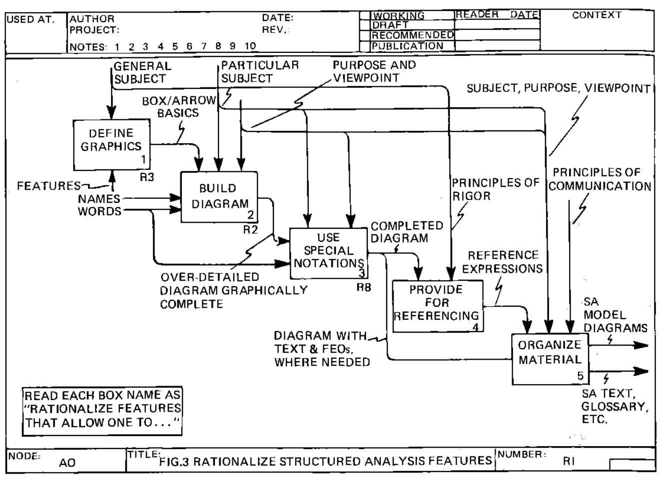


FIGURE 16. SA diagram [*64*].

designed is based. It is assumed that a different processor is available and used to execute any needed action or data transformation, so no resource-sharing policies have to be considered in this phase; they are considered later in the implementation (software design) phase.

SSD diagrams, like all JSD diagrams, can be transformed easily to pseudocode, dealing with all actions on entities.

Functions are added to SSDs in the next step of the methodology, indicating how to obtain output from information represented in the model. An example is presented in Figure 19, where inquiries can be made as to information available on bank customers, described previously.

The last step of JSD, not relevant for our work in logical design, is the transformation of SSDs into system implementation diagrams (SIDs), where more realistic hypotheses are done on the number of available processors and time constraints are taken into consideration.

6.0 ENVIRONMENTS FOR REQUIREMENTS SPECIFICATION

In this section some computer-supported environments for system design are presented. Section 6.1 discusses support environments for OIS design; section 6.2 reviews software engineering environments to support the development of various types of software systems.

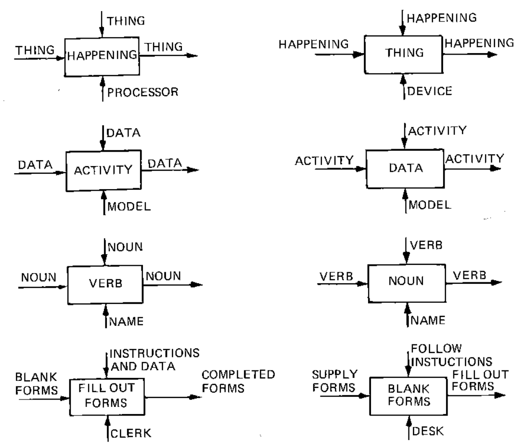


FIGURE 17. Duality of activities and data in SADT [*64*].

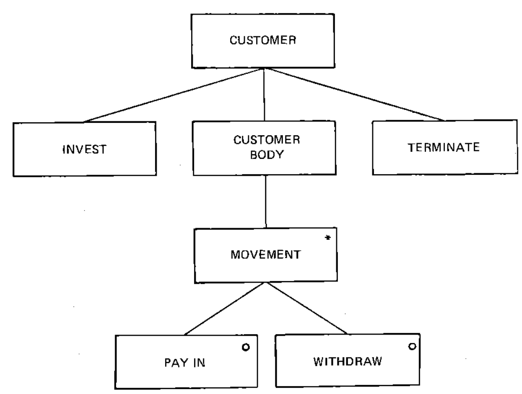


FIGURE 18. JSD structure diagram [*65*].

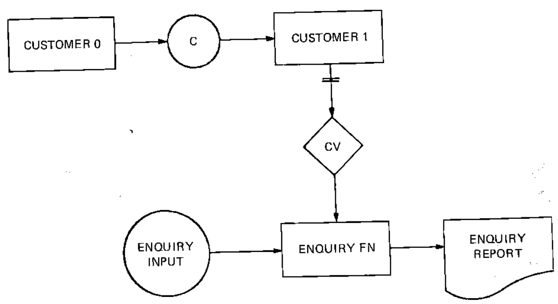


FIGURE 19. JSD functions in SSD diagrams [65].

6.1 Office Environments

Two systems to support OIS design are presented in this section: the OFFIS design methodology, supported by a specification and analysis system, and the Quinault editor for ICN graphs.

6.1.1 OFFIS

OFFIS is a system designed to facilitate an interactive and iterative office analysis and design process, providing the planner/designer of the automated office with a flexible method of analyzing system features and constraints [66]. OFFIS has been developed at the Department of Management Information Systems of the University of Arizona.

The system is based on the OFFIS model for an office. The elements of the model can be specified in the OFFIS language by planners/designers, who are part of the office personnel, or managers. The language has a simple syntax and is user friendly and nonprocedural.

In the specification of office requirements, the system is divided into sections describing categories in the office (e.g., departments or types of documents). The elements specified with statements of the language describe requirements and constraints of the office. These specifications lead to an incremental construction of the office model. Besides model construction, the OFFIS system also provides the possibility of analyzing the consistency and completeness of the office system under examination through a set of rules and obtaining corresponding reports. The specifications are collected in the OFFIS database and used by the analyst (see Fig. 20).

The requirements analysis is based on a technical approach to requirements collection. Basic elements in an OFFIS model are objects, attributes, and relations. Objects are static elements, which describe data and agents, seen as a collection of data. Attributes are related to objects. Relations describe relationships among objects, and their main purpose is to express possible operations on objects. Relations can also express conditions, durations, and hierarchical organizational relations, in terms of persons that must report to other persons.

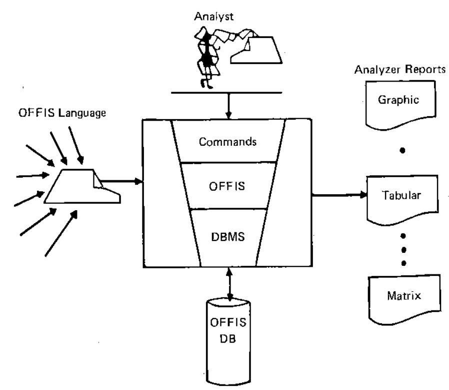


FIGURE 20. Overview of the OFFIS system [66].

The information flow is expressed through the relations, providing a weight-to-measure importance of relations.

The OFFIS methodology is based principally on a technical model of office reality, and office elements are studied in great detail. However, it takes into consideration some of the organizational divisions in the office, thus enriching the technical approach to the design.

Several design tools are provided, mainly for the analysis of the consistency and correctness of the conceptual model resulting from users' requirements specification. Some specifications of objects using the OFFIS language are shown in Figure 21.

6.1.2 Quinault

Quinault [67] is an experimental system for modeling and analyzing offices and OIS, based on an extended version of ICNs.

The ICN method of office modeling was presented in Section 3.1.2. ICNs are graph models of data and control flow in offices, composed of a set of activities and a set of repositories. Each activity corresponds to the informal notion of a task, whereas each repository represents some storage medium. An activity can retrieve data from a repository and insert data into a repository for storage. The data reference is represented by an arc between a repository node and the activity node; activities are interconnected by another class of arcs, which represent the precedence constraints on the set of activities. Arc labels specify the identity of information that flows between repositories and activities and are used to analyze the data flow characteristics of the model. Each activity in the model can have information to specify the time required to execute that activity and possibly, a procedural interpretation of the function of the activity.

```
JOB TITLE OFFICE SECRETARY:
 CREATES QUARTERLY MEETING NOTICES;
 SCHEDULES QUARTERLY MEETING
  USING P CALENDAR, VP CALENDAR;

MEETING QUARTERLY MEETING;
 HAPPENS 4 TIMES YEARLY;
 ATTENDED BY PRESIDENT, V PRESIDENT,
  ACCOUNTANTS;
DURATION 2 HOURS:

DATA FILE QUARTERLY MEETING FILE:
 UPDATED BY OFFICE SECRETARY
  USING QUARTERLY MEETING NOTICE:

LETTER QUARTERLY MEETING NOTICE;
 HAPPENS 4 TIMES YEARLY;
 SENT TO ACCOUNTANTS
 ROUTED TO P MAILBOX, VP MAILBOX,
  EMPLOYEE MAILBOX:

EXTERNAL NAME ACCOUNTANTS;
```

FIGURE 21. The OFFIS language [67].

A particular ICN is not expected to be the only representation of the system. Instead, an ICN can be thought of as a view of the system at some level of abstraction. It is possible to use hierarchical refinement of the activities in the model to specify the details of an activity in terms of another ICN model. Figure 22 shows a graph of an ICN model for filling out an income tax return.

ICN models can serve as descriptive tools to document data and control flow. In addition, ICNs can help determine certain structural properties of the office and specify alternate processing activities, for example, reducing the need for file access for maximizing concurrent processing. The graph models can also serve as a basis for analyzing the flow for anomalous situations by applying consistency algorithms (e.g., deadlock detection).

Quinault integrates many features of ICN models in a highly interactive personal computer system (see Fig. 23).

The editor subsystem is used to construct an annotated ICN graph that is permanently stored in a file denoted "model.icn." The editor provides three different menus: The first is used to draw and erase nodes and arcs in the graphics window and implicitly, to enter description into the data base, whereas the second contains commands to annotate the model. The third menu contains commands to perform all other forms of model manipulation within the editor's capabilities.

The analyzer can be used statically to analyze the description and to compute optimal ICN transformations of the original representation; a transformed ICN is written into a new file in the internal format produced by the editor. For example, ICN can be used as the basis of a queuing network model from which statistical properties of the transaction processing rates in the office can be determined; the results of that analysis can then lead to model refinements.

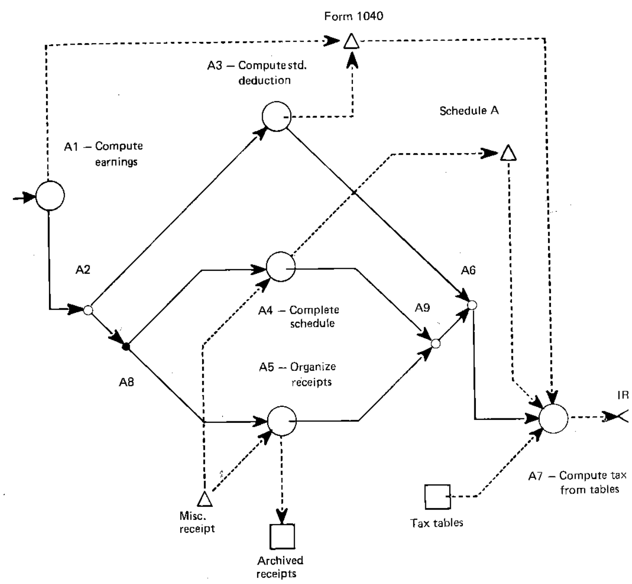


FIGURE 22. Graph for an ICN model [67].

The Quinault simulator uses the ICN description to animate transaction flow through an ICN in scaled real time. It can be used to inspect the graph model quickly to find bottlenecks or other performance problems.

A prototypal version of Quinault has been implemented on the Xerox/Alto a 16-bit microprogrammed minicomputer in the Mesa programming environment an example of its performance is illustrated by Nutt and Ricci [67].

6.2 Software Engineering Environments

In this section, software engineering environments are presented for the development of software products or information systems. In many aspects, OIS and software development techniques are similar, particularly in the earlier design phases, where a conceptual representation of the system has to be given. Therefore, as in Section 5.0, where some general requirements specification methodologies were presented here, some tools are described that have been developed for supporting the first phases of system design. The tools have been created for supporting SA (described in Section 5.1),

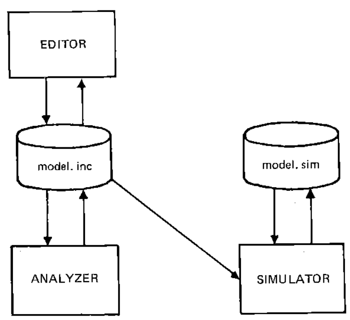


FIGURE 23. Quinault configuration [67].

for supporting formal requirements specifications and analysis (PSL/PSA), for supporting the concept of Ada programming environment and for supporting data base design with the entity–relationship model.

6.2.1 SA

SA is a specification methodology for engineering and information systems design [63] described in Section 5.1. The first step of the methodology consists in examining information that flows into and out of the system being modeled; the second determines transformations made on this information as if flows from input to output. An important aspect of SA is that it is hierarchical in nature: As portions of the system are identified, they themselves can be modeled as distinct systems. This allows understanding of the problem in terms of components that are simple enough to be understood easily.

SA produces three types of documents. First, there is a hierarchy of DFDs, where each DFD describes a system that has distinct data flowing in and transformations on that data that eventually change it into the output data. Each transformation is detailed either with another DFD or with a "minispec." A minispec, the second type of document, is a description of a problem that is simple enough to be stated in a restricted form of English called structured English. The third type of document, the data dictionary (DD) is a collection of definitions of the data items used in the DFDs and minispecs.

A system to support these types of documents has been developed at Tektronix [68]. Figures 24 through 27 present the graphic and textual representation produced by that system for an example of specifications of an automatic teller machine.

SA has been used successfully at Tektronix for several years. Use of SA in the Tektronix engineering environment was analyzed to determine what functions should be automated; SA itself was used to perform this analysis. As a result, SA support tools, written in Modula-2 under UNIX V7, have been developed to include the following capabilities:

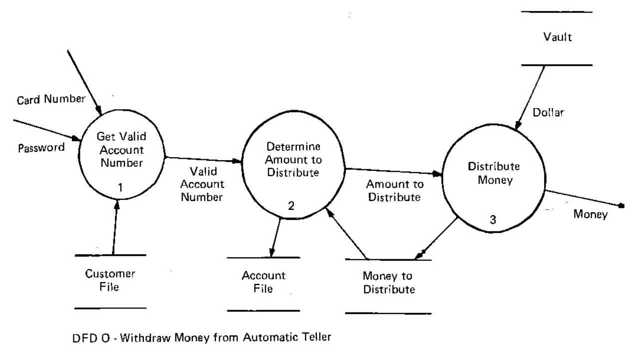


FIGURE 24. Top-level DFD for the automatic teller example [68].

- Modify and display SA documents.
- Check for errors in SA documents, for example, an inconsistent use of a data flow among several hierarchically related DFDs.
- Help maintain the consistency of the SA documents.

Four families of support tools exist that can be used individually or in combination. The Edit tool modifies the SA document interactively and derives change requests that help maintain the consistency of the SA documents the Evaluate tool detects errors in an SA document; the Format tool converts the internal representation of an SA document into a form that is suitable for viewing; finally, the Cleanup tool helps maintain the consistency of the SA

Account-File = {Account-Number + Balance-in-Account}
Account-Number = ID-number
Amount-to-Distribute = number-of-dollars
Amount-to-Withdraw = number-of-dollars
Balance-in-Account = number-of-dollars
Card-Number = ID-number
Customer-File = {Card-Number + Password + Account-Number}
Money = {dollar}
Money-to-Distribute = number-of-dollars
Password = ID-number
Valid-Account-Number = ID-number
Valid-Amount-to-Withdraw = number-of-dollars
Vault = {dollars}

FIGURE 25. Data dictionary for the automatic teller example [68].

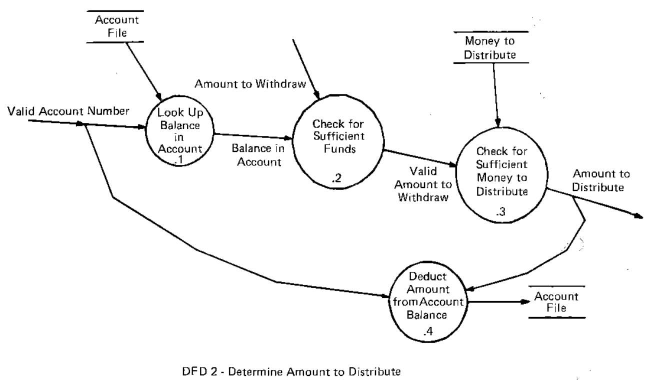


FIGURE 26. Second-level DFD for the automatic teller example [68].

documents by validating and performing changes that are derived by the Edit tool. Figure 28 illustrates how SA support tools can work together.

A Computer-Aided SA tool (PCSA) for the IBM PC/XT/AT has also been developed and commercialized recently by StructSoft, Inc.

6.2.2 PSL/PSA

PSL/PSA is a computer-aided structured documentation and analysis technique that was developed, and is being used for, analysis and documentation of requirements and preparation of functional specifications for information processing systems [69].

2.2 Policy to Check for Sufficient Funds

Check to see if the Amount-to-Withdraw is less than or equal to the Balance-in-Account.
If Amount-to-Withdraw is less than or equal to the Balance-in-Account then:
Valid-Amount-to-Withdraw becomes the Amount-to-Withdraw
otherwise:
Report an error.

FIGURE 27. Sample minispec for the automatic teller example [68].

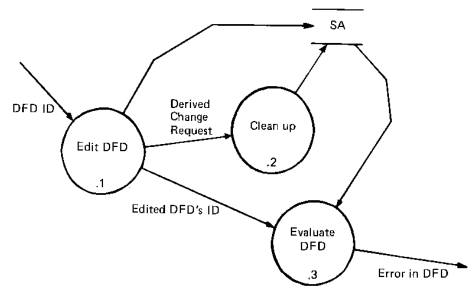


FIGURE 28. Edit/Cleanup/Evaluate Combination [68].

IS development procedure requires that the proposed system be reviewed before a major investment is made in system construction. On the basis of this review, it may be decided to proceed with the physical design and implementation, to revise the proposed system, or to terminate the project. The review is usually based on a document proposed by the project team that contains various types of information, such as the system organization, a description of its operation to verify whether it accomplishes the proposed objectives and a cost/benefit analysis and recommendation.

In a computer-aided logical design system such as PSL/PSA, the objective is, as in the manual process, to produce system definition reports for analysis and evaluation. The capability of describing systems in computer processible form is provided by the use of the system description language called PSL, which is based on a model for information systems. The model consists of objects that may have properties and the objects may then be connected or interrelated in various ways; these connections are called relationships.

The ability to record such description in a database, incrementally modify it, and perform analysis and produce reports on demand comes from the software package called Problem Statement Analyzer (PSA). The analyzer is then controlled by a command language (see Fig. 29).

The use of PSL/PSA does not depend on any particular structure of the system development process or any standards on the format and content of hard copy documentation. Using the system, the data collected or developed during the phases of the logical system design process are recorded in the database. These data can be analyzed by computer programs, and intermediate documentation can be prepared on request. These reports provide various types of analysis and summary on the information in the database, for example, data flows in a graphic form, also allowing error correction and recovery.

Many similarities may be found between PSL/PSA and the OFFIS system described in Section 6.1.1. They are based on the same concepts, but in the case of OFFIS, the specification language for the system is tailored to the office environment.

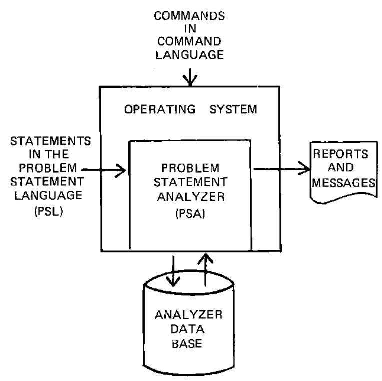


FIGURE 29. PSA [69].

ISDOS, Inc., has commercialized a PSL/PSA system that runs on most medium and large mainframe operating systems including VAX/VMS and UNIX and workstations such as Pixel and Apollo. Furthermore, a new product called Structured Architect, supporting IBM-PC, PC/XT, AT, integrates comprehensive analysis and graphics for SA, providing facilities to combine the work of multiple workstations through PSL/PSA on a host system, with the ability to handle large multianalyst projects easily.

6.2.3 APSE

The APSE concept was first described in [70]. An APSE is a system designed to support the development and maintenance of large-scale software systems written in Ada throughout the whole life cycle. APSE is intended to be oriented specifically toward software for embedded computer applications.

Tools to form an APSE, giving coherent support for the whole software life cycle, can be identified once a coherent software development and maintenance methodology has been produced. A life-cycle model is a model of the full lifetime of a system, from initial conception to final obsolescence. It should show the end products of the system development and how these end products are derived and verified.

To master the complexity of the system under development, it is necessary to produce a series of descriptions at different levels of abstraction, following a set of rules that govern the transformations by which one representation is produced from another. A coherent APSE is thus based on a set of methods that cover every aspect of system development and maintenance according to a particular interpretation of the life-cycle model. Specifically, there should be no part of the development process that has to be pursued without the guidance of a particular rule; furthermore, verification procedures

should be concerned with assessing the consistency and completeness of the representations and showing the accuracy of the transformations from the other representations as far as possible.

The description of a particular type of coherent methodology, as well as tools and data base facilities that would be necessary to support that methodology, is given in Ref. 71. The resulting framework for a coherent APSE is composed of the following levels, as illustrated in Figure 30:

- Level 0: Hardware and host software as appropriate.
- Level 1: Kernel APSE (KAPSE), which provides database, communication, and run-time functions to enable the execution of an Ada program (including an APSE tool) and which presents a machine-independent portability interface.
- Level 2: Minimal APSE (MAPSE), which provides a minimal set of tools that are both necessary and sufficient for the development and continuing support of Ada programs. These tools will be written in ADA and will be supported by KAPSE.
- Level 3: APSEs that are constructed by extensions of the MAPSE to provide fuller support of particular applications or methodologies.

APSEs currently available or under development are listed in Table 1.

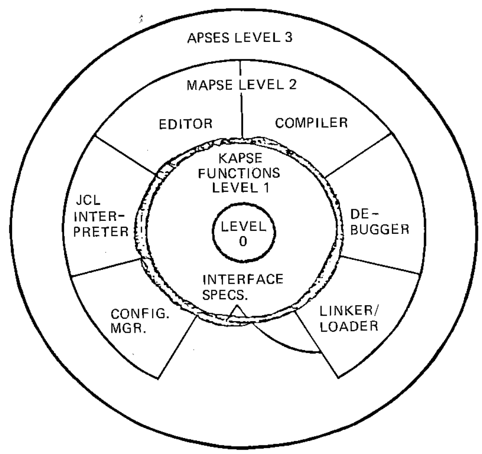


FIGURE 30. Framework of a coherent APSE.

TABLE 1 APSES

ADE	ROLM	MSE/800
ALS	SofTech	VAX/VMS
AIE	Intermetrics	IBM 370/UTS
SPERBER	Mod-RFG	Siemens
PAPS	CEE	VAX/UNIX-M40
Spice	CMU	Perq
Arcturus	Univ. Irvine	VAX/UNIX
AISS	PFI-CNR	M40

6.2.4 INCOD

INCOD (Interactive Conceptual Design) [72] is a system for conceptual data base design developed as part of the research activity carried on within the subproject DATAID of PFI, a project on applied computer science sponsored by the Italian National Research Council. A prototype of the proposed system has also been implemented in a DM-IV environment supported by a Honeywell DPS8/44 computer.

In the data base design process, the conceptual step has the goal of producing a complete and implementation-independent description of the information of interest for the application. This specification must be formal and based on a semantical model, that is, a model that provides a set of constructs for the explicit representation of the semantics of data and description of their dynamic properties. The complexity of the task of conceptual design has led to the acknowledgment of the importance of automated tools that enrich the capacity of the designer, especially when large applications have to be tackled, interactively guiding the designer through the steps of some activities of the design. By handling data concerning that to be stored in the data base (metadata), automated systems can verify their mutual consistency and simplify the management of the corresponding documentation.

INCOD allows the interactive stepwise definition of the static and dynamic requirements, possibly subdivided in various views, the integration of the views, the continuous check of the consistency of the design process, and the organization of the documentation.

The static aspects are described by means of a data schema, based on an extended version of the Entity–Relationship (E-R) model, which includes the useful concepts of subset relationship and generalization hierarchy. The dynamic aspects are instead described through a set of transactions, which model the operations to be performed on the database, and a set of events, which take into account the notion of time and mutual and casual relationships among operations. Transactions are expressed in an E-R-specific language, suitably defined, and events in a modified version of Petri nets.

The type of interaction with the user implemented in the prototype is based on the principle that the system guides the user in the correct and complete use of the available functions. The tool supports a set of functions arranged in a hierarchical structure (see Fig. 31): functions are useful both in the phases of design and maintenance, because they perform aid in the definition of data, automatic check of data consistency, and efficient management of documentation.

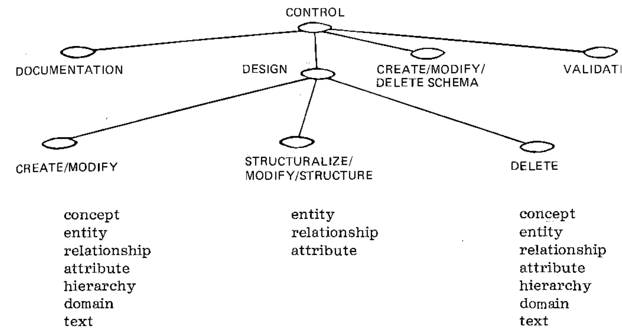


FIGURE 31. Hierarchical structure of functions.

It is possible to distinguish between intermediate functions (allowing access to other functions) and primitives (performing transformations on the data). When the user interacts with an intermediate function, the system displays the list of the functions that can be accessed. When the user interacts with a primitive, the system displays the list of the data that can be defined or modified by means of forms. Each form contains a set of graphic characters that delimit the field in which the user can insert the values and the phrases that explain the meaning of the fields. If the data filled in the received form do not permit the function to perform its task, the function sends a diagnostic message: The user may try to finish the function interactively, modifying the form data and sending them again.

A powerful graphic interface allows the user to have a synthetic view of the produced schema and to update the schema itself, simulating the actions the designer once performed with paper, pencil, and eraser. An example of a diagram produced by the tool is illustrated in Figure 32.

6.2.5 RAMATIC

RAMATIC is a computer graphics-based tool under development by the System Development Laboratories of Gothenburg (SYSLAB-G) as part of the Program in Information Processing launched by the Swedish National Board for Technical Development (STU) [73–77].

The main purpose of this tool is to support ongoing discussions of a project group in real time. RAMATIC supports systems analysis by supplying several model types. Many systems analysis and design approaches only supply one model type, and most techniques can only describe a few aspects of an organization or an information system; on the other hand, a set of different model types can describe several interesting aspects in connection with systems development. This is why the tool allows each group involved in the project to adopt its own description technique for focusing on a particular slice of reality and then integrating related descriptions.

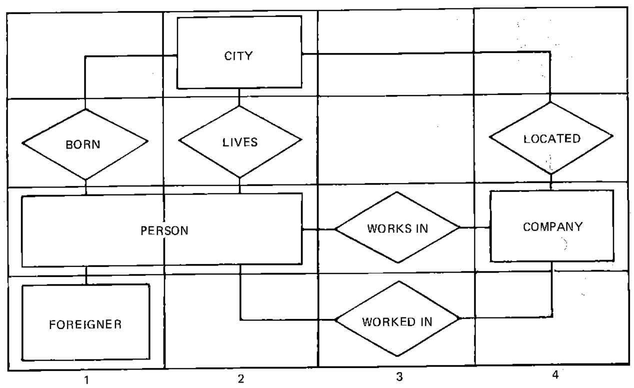


FIGURE 32. Diagram produced by the INCOD graphic tool [72].

RAMATIC is so general that previously unknown description techniques can be entered into the tool; the tool then supports the syntax for the graphics rules governing its use, and rules for the use of interrelated description techniques. A basic assumption underlying RAMATIC developments is that the possibility of expressing structures in graphic form contributes to better comcunication. Thus, the tool provides several facilities that allow the user to sketch, draw, and change graphs directly on the screen with a mouse-controlled cursor. The pictures and their semantic interpretation are then stored with a binary relational database system. However, RAMATIC is not limited only to graphic input and output. One of its interesting features is that from a given specification of a system with its node and relationships expressed in a special specification language (in textual form), the tool can generate a suitable picture on the screen automatically. This facility allows the creation of diagrams through symbolic design, resulting in a conceptual representation as graphic design.

RAMATIC is internally separated into four functions, as can be seen in Figure 33.

The application program (AP) of the RAMATIC stores and retrieves the graphs of the different modules drawn on the graphic terminals with the aid of the application data structure (ADS). ADS consists of both a CS4 database structure, which is used for storing different model type descriptions and menu layouts for command windows, and a set of subroutines for communications with the database handler. In the CS4 database of the RAMATIC tool, information on pictures and symbols, for example, rectangles or rhomboids, is stored in the Symbol Data Base (SymDB) whereas the Spatial Data Base (SDB) contains the spatial representation of the pictures. In addition, the Conceptual Data Base (CDB) retains information on the semantic description of node types and what relationships between these node types are allowed.

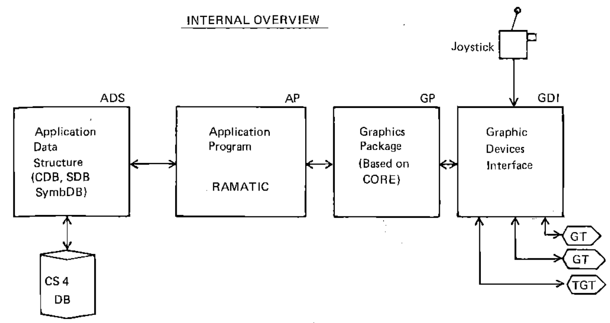


FIGURE 33. The internal functions of RAMATIC [73].

To interact with the users at the terminal, AP uses subroutines in a graph ic package (GP) for both pictures and ordinary texts. Finally, between the users and the GP, there is a graphic device interface (GDI).

When a model type has a syntax description associated with it, it is impossible to create pictures that violate the syntax. The RAMATIC tool supports analysis of qualitative aspects of models, for example, the design of correct models. As a modeling support system, RAMATIC uses three types of representation: (*a*) Spatial representation focuses on the physical content of diagrams; (*b*) conceptual representation defines the semantical and topological content of diagrams; and (*c*) to make analyses of correctness of modules, ther is a need for explicit specification of a modeling approach, as well as knowledg on the design process itself.

Knowledge of a specific type of model is represented in first-order predicate logic and the conceptual representation of a model, that is, the data stored in the CDB, as a representation of the theory. Therefore, during the evolution of the CDB, a test for satisfiability is performed to determine whethe a state is a model to the theory.

6.3 Project Management Tools

6.3.1 XCP

XCP is an experimental tool that assists office workers in coordinating their actions in space and time according to predefined office procedures. A prototypal version of the tool has been implemented in the VAX LISP language on VAX 11/785, 11/780, and 11/750 processors under the VMS operating system by the Intelligent System Technologies Group of Digital Equipment Corporation [78].

There are several key concepts that XCP implements. A person is simply a human being. A role is the codification of some task. Most people assume several roles during a working day, such as project leader, product manager, author, reviewer, and so on. In an office procedure, each role is responsible

for some part of the total activity and carries with it a set of rights, responsibilities, and expected behaviors with respect to other roles. An actor is a person who has assumed a role. A document is the symbolic representation of some paper form, for example, an order, a request for a proposal, a software problem report.

XCP allows its users to define protocols, which are plans of cooperative activities. A protocol defines the task of which an office procedure is comprised. It coordinates the actions of the office staff and supports them in carrying out the office procedures. XCP tracks tasks through the defined protocols, keeps a history of all tasks on a per-document basis, and informs users of the status of each task. A user may suspend the protocol at any point during the protocol and resume it later at the same point. It is important to note that XCP is a general coordinator program and is not tied to any particular application. It is the specific applications that tailor its general capabilities to a specific purpose. Protocols are entered into the system through a set of simple commands; while querying XCP, one can obtain information on the state of documents, on people's roles, and on protocols.

Figure 34 illustrates the architecture of XCP, reflecting the division of functionality within the system. The terminal specific part of XCP is called the command interface (CI). It is logically and physically a separate program from the terminal independent part, which is referred to as the Protocol Task (PT). The CI acts as the user front end and interacts directly with a person at a terminal; there is one CI per active user. Each CI interacts with a single PT through a variety of messages sent via operating-system-supported buffers. These messages represent either user-instigated commands or choices made in response to demands from the PT as it executes protocols. An example of XCP usage, employing an order-processing protocol, is given in Figure 35.

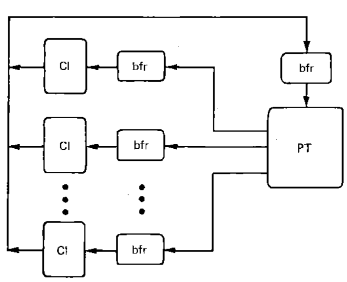


FIGURE 34. The architecture of XCP [78].

XCP output	user input	comments
XCP Your name? Waters		Start XCP from VMS operating system and identify WATERS as a person. Persons can only log in, assume roles and log out.
You can assume these roles: ADMIN SHIPPER Your choice? A		Presumably another role-player has given WATERS permission to assume these roles.
XCP> show person Waters		Show all documents which Waters "ow in any role.
You are responsible for the following documents: Title — Role ORD15 Order from Urbane Company — ADMIN ACK14 Acknowledgement for ORD15 — ADMIN DONE25 Done-notice for ORD7 — ADMIN ORD22 Order from Jones Ltd — SHIPPER SHIP Shipped-notice for ORD5 — SHIPPER		There are four types of documents of interest here. Orders represent a a customer request for goods. Ackno edgements assure the CLERK entering the order that it will be handled by SHIPPERs. Shipped-notices assure th ADMIN that the goods were shipped. Done-notices report the shipment to t CLERK who originally entered the or
XCP> status ORD15 The last thing you did was attach ORD15 to ACK14 The next thing you can do is dispatch ACK14 to BARNES@CLERK.		XCP knows where the protocol for ORD15 was interrupted, and that AC must be handled next as part of the protocol for ORD15. WATERS worked on ORD15 and suspended the protoco after creating ACK14 (but before dispatching it).
XCP> work-on ACK14		Resume the protocol for ACK14 by re starting it at the point it was interrupted.
Now you can dispatch ACK14 to BARNES@CLERK. First you can enter text to be sent. Your options are: text <filename> ctl-Z ? Your choice? stdack.txt		
Your options are: text <filename> ctl-Z ? Your choice? ctl-Z		XCP continues to ask for more text until WATERS signals that there is n by typing "control-Z".
Should ACK14 be dispatched to BARNES @CLERK now? Your options are: yes no ctl-Z ? Your choice? yes		WATERS dispatches the acknowledgem to the originating CLERK. If WATER had replied "no" or "control-Z" to th dispatch question, the protocol would have been suspended at that point u WATERS resumed it by giving the co mand to work-on ACK14.
ACK14 has been dispatched to BARNES@ CLERK. You have completed your work on ACK14.		

FIGURE 35. An example of SCP usage [35]. (continued)

XCP output	user input	comments
Now you can dispatch ORD15 to someone in SHIPPER. First, you can enter text to be sent.		Recall that this protocol required an acknowledgement to be sent to the originating CLERK before the order is forwarded to a SHIPPER. Since that obligation has been fulfilled, the protocol now continues by prompting WATERS to assign the order to a SHIPPER.
Your options are: text <filename> ctl-Z ? Your choice? text Type your text and end it with control-Z. This order should be shipped as soon as possible. ctl-Z		
Your options are: text <clientname> ctl-Z ? Your choice? ctl-Z		
Next step is to dispatch ORD15 to someone in SHIPPER. Your options are: <person> anyone ctl-Z ? Your choice? STONE		WATERS sends it to the person STONE in the role of SHIPPER. Had it been sent to the role but to no specific person, presumably someone in the role would take charge of it.
ORD15 has been dispatched to STONE@ SHIPPER.		
XCP> status ORD15		WATERS has done all work possible on both ACK14 and ORD15. STONE@SHIPPER will eventually send a shipped-notice for ORD15 and WATERS must then send out a done-notice to BARNES@CLERK, the originating CLERK. Now WATERS can ask XCP to work on some other document, for status information, to assume another role, or to log out.
The first thing you did was dispatch ORD15 to STONE@SHIPPER. You are awaiting a reply from STONE@ SHIPPER.		
XCP> logout		
Ending XCP		

FIGURE 35 (Continued)

7.0 CONCLUDING REMARKS

In this article a survey of work in the literature has been presented that can be used as a basis for the development of a design support tool for logical design of an office system. Office characteristics, a classification of existing office models, a description of office design methodologies, and office design support environments have been reviewed, along with some extensively applied general system development methods and design support environments.

ACKNOWLEDGMENTS

This work has been supported by Politecnico di Milano and by the Italian National Research Council (CSISEI-CNR) at Politecnico di Milano.

REFERENCES

1. M. Hammer and J. Kunin, "Design Principles of an Office Specification Language," in *Proceedings of the AFIPS National Computer Conference* (May 1980), pp. 541–547.
2. J. Mylopoulos, P.A. Bernstein, and H.K.T. Wong, "A Language Facility for Designing Database-Intensive Applications," *ACM Trans. Database Syst.*, 5 (2) (June 1980).
3. M. Sirbu, S. Schoichet, J. Kunin, and M. Hammer, "OAM: An Office Analysis Methodology," MIT Report, OAM-016, Cambridge, MA, 1981.
4. M.D. Zisman, "Use of Production Systems for Modeling Asynchronous, Concurrent Processes," in *Pattern Directed Inference Systems*, Academic Press, New York, 1978.
5. M.M. Zloof, "QBE/OBE: A Language for Office and Business Automation," *Computer* (May 1981).
6. N. Naffah, ed., *Integrated Office Systems—Burotics*, North-Holland, Amsterdam, 1980.
7. N. Naffah, ed., *Office Information Systsms*, North-Holland, Amsterdam, 1982.
8. T. Malone, "How Do People Organize Their Desks? Implications for the Design of Office Information Systems," *ACM Trans. Office Inf. Syst.*, 1 (1), 99–112 (January 1983).
9. D. C. Tsichritzis, "Form Management," *Commun. ACM*, 25(7) (1982).
10. P. Economopoulos and F. H. Lochovsky, "A System for Managing Image Data," in *Proceedings of the IFIP '83*, Paris, September 1983.
11. K. Tabata, T. Machida, H. Takeda, and H. Kambayashi, "High-Speed Image Scaling for Integrated Document Management," *ACM-SIGOA Conference on Office Information Systems*, Toronto, 1984.
12. D. Tsichritzis and S. Christodoulakis, "Message Files," *ACM Trans. Office Inf. Syst.* 1(1) (January 1983).
13. A. Wegman, "VITRAIL: A Window Manager for an Office Information System," *ACM-SIGOA Conference on Office Information Systems*, Toronto, June 1984.
14. W. Horak and B. Kronert, "An Object-Oriented Office Document Architecture Modeling for Processing," *ACM-SIGOA Conference on Office Information Systems*, Toronto, June 1984.
15. J.D. Gould and S.J. Boeis, "Human Factors Challenges in Creating a Principal Support Office System—The Speech Filing System Approach," *ACM Trans. Office Inf. Syst.*, 1(4) 273–298 (October 1983).
16. N. Meyrowitz and A. van Dam, "Interactive Editing Systems: Part I and II," *ACM Comp. Surv.* 14(3), pp. 321–416 (September 1982).
17. R. Furuta, J. Scofield, and A. Shaw, "Document Formatting Systems: Survey, Concepts, and Issues", *ACM Comput. Surv.*, 14(3) 417–472 (September 1982).
18. D. Tsichritzis, "Tools for Office Systems," *Proceedings of IFIP TC8 Conference on Office Systems*, Helsinki, September 1985.
19. F. Barbic and S. Illuzzi, "An Office Message System," in *Proceedings of RIAO '85*, Grenoble, March 1985.
20. W.B. Croft and R.H. Thompson, "The Use of Adaptive Search Strategies in Document Retrieval Systems," in *Research and Development in Information Retrieval* (C.J. van Rijsbergen, ed.), 1984, pp. 95–111.
21. P.A. Larson, "A Method for Speeding Up Text Retrieval," *ACM Sigmod Database Week*, San Jose, CA, 1983.

22. G. Sacco, "OTTER—An Information Retrieval System for Office Automation," *ACM-SIGOA Conference on Office Information Systems*, Toronto, June 1984.
23. S. Christodoulakis, "Issues in the Architecture of a Document Archiver using Optical Disk Technology," in Proceedings of the *SIGMOD '85 Conference*, Austin, TX, 1985.
24. W. Horak, "Office Document Architecture and Office Document Interchange Formats: Current Status of International Standardization," *Computer*, 50–60 (October 1985).
25. G. Kroenert, J. Freidrich, U. Schneider, and G. Lauber, "Requirements on a Formatting Document Editor Based on the ECMA Standard 101," in *ESPRIT Technical Week '85*, Brussels, September 1985.
26. S. Ceri and G. Pelagatti, eds, *Distributed Data Bases: Principles and Systems*, Mc Graw-Hill, New York, 1984.
27. Diener et al., "Replicating and Allocating Data in Distributed Data Systems for Workstations," in *1985 ACM Sigsmall Symposium on Small Systems*, Danvers, MA, May 1985.
28. G. Bracchi and B. Pernici, "The Design Requirements of Office Systems," *ACM Trans. Office Inf. Syst.*, 2(2), 151–170 (April 1984).
29. D.D. Chamberlin, O.P. Bertrand, M.J. Goodfellow, J.C. King, D.R. Slutz, S.J.P. Todd, and B.W. Wade, "JANUS: An Interactive Document Formatter Based on Declarative Tags," *IBM Syst. J.*, 21(3) (1982).
30. M.M. Zloof, "Office-By-Example: A Business Language that Unifies Data and Word Processing and Electronic Mail," *IBM Syst. J.*, 21(3) (1982).
31. F. Barbic and B. Pernici, "Time Modeling in Office Information Systems," in *Proceedings of the SIGMOD Conference*, Austin, TX, May 1985.
32. R. Maiocchi, and B. Pernici, "Time Reasoning in the Office Environment," Proceedings of the IFIP 84 Conference, Pisa, Italy, October 1986.
33. R.C. Harkness, "Office Information Systems: An Overview and Agenda for Public Policy Research," *Telecommun. Policy* (June 1978).
34. P.A. Wilson and J.A.T. Pritchard, *Office Technology Benefits*, National Computing Centre, Manchester, England, 1982.
35. D.W. Conrath, C.A. Higgings, R.H. Irving, and C.S. Thachenkary, "Determining the Needs for Office Automation: Methods and Results," *Office, Tech. and People* (May 1983).
36. P.A. Strassmann, "Overview of Strategic Aspects of Information Management," *Office, Tech. and People* (March 1982).
37. F.H. Lochovsky, "A Knowledge-Based Approach to Support Office Work," in *Data Base Engineering*, W. Kim, D. Ries, F.H. Lochovsky, eds., 2, 43–51, IEEE Computer Society Press, New York, 1983.
38. D. Tsichritzis, ed. "Objectworld," in *Office Automation*, Springer-Verlag, 1985.
39. J. Hogg, M. Mazer, S. Gamvrouls, and D. Tsichritzis, "Imail—An Intelligent Mail System," in *Data Base Engineering*, W. Kim, D. Ries, F.H. Lochovsky, eds., IEEE Computer Society Press 1983, Vol. 2, pp. 167–182.
40. C.A. Ellis, R. Gibbons, and P. Morris, "Office Streamlining," in *Integrated Office Systems*, (N. Naffah, ed.), 1980.
41. G. Barber, "Supporting Organizational Problem Solving with a Workstation," *ACM Trans. Office Inf. Sys.* 1(1) (January 1983).

42. M.M. Zloof P. De Jong, "The System for Business Automation (SBA): Programm," *Commun. ACM*, 20(6) (June 1977).
43. J. Dunnion, D.J. Harper, B. Anker-Moeller, T. Bogh, M. Sherwood-Smith, and C.J. Van Rijsbergen, "Minstrel-ODM: A Basic Office Data Model," in *ESPRIT Technical Week '85*, Brussells, September 1985.
44. D.W. Shipman, "The Functional Data Model and the Data Language DAPLEX," *ACM Trans. Database Syst.*, 6(1), 140–173 (1981).
45. C.A. Ellis and M. Bernal, "Office Information System," in *Conference on Office Information Systems*, Philadelphia, PA, June 1982, pp. 131–141.
46. C. Cook, "Streamlining Office Procedures—An Analysis using the Information Control Net Model," in *Proceedings of the AFIPS National Computer Conference*, May 1980, pp. 555–565.
47. W.B. Croft and L.S. Lefkowitz, "Task Support in an Office System," *ACM Trans. Office Inf. Syst.*, 2(3), 197–212 (July 1984).
48. A. Goldberg, "Introducing the Smalltalk-80 System," *Byte*, 6(8), (August 1981).
49. M. Ahlsen, A. Bjornerstedt, S. Briffs, C. Hulten, and C. Soderlund, "An Architecture for Object Management in OIS," *ACM Trans. Office Inf. Syst.*, 2(3) (July 1984).
50. G. Bracchi and B. Pernici, "SOS: A Conceptual Model for Office Information Systems," *Data Base*, 15(2) (Winter 1984).
51. S. Sarin and I. Greif, "Software for Interactive On-Line Conferences," *Conference On Office Information Systems*, Toronto, June 1984, pp. 46–58.
52. K. Kumano, Y. Nagai, and M. Hattori, "MACROS: An Office Application Generator," in *Management and Office Information Systems*, (S-K. Chang, ed.), Plenum Press, New York and London, 1982, pp. 369-384.
53. B. Croft and V. Lesser, "Progress Report: POISE Project Intelligent Interfaces for Task-Based Systems," University of Massachusetts Report, Amherst, December 1984.
54. J.A. Spriet and G.C. Vansteenkiste, *Computer-Aided Modelling and Simulation*, Academic Press, New York, 1982.
55. P. Dumas, G. du Roure, C. Zanetti, D. Conrath, and J. Mairet, "MOBILE–Borotique: Prospects for the Future," in *Office Information Systems*, (N. Naffah, ed.), North-Holland Publishing Co., Amsterdam, 1982, pp. 471–480.
56. F. Barbic, S. Ceri, G. Bracchi, and P. Mostacci, "Modeling and Integrating Procedures in Office Information Systems Design," *Inf. Syst.*, 10(2), 149–168 (1985).
57. P. Henderson, "Functional Programming, Formal Specification, and Rapid Prototyping," *IEEE Trans. Software Eng.*, (SE-12), 2 (February 1986).
58. W. Hershey, "Idea Processors," *Byte*, (10),6 (1985).
59. R.R. Panko, "38 Offices: Analyzing the Needs in Individual Offices," *ACM Trans. Office Syst.*, (2), 3 (July 1984).
60. C. Floyd, "A Systematic Look at Prototyping," in *Approaches to Prototyping*, (R. Budde *et al.*, eds.), Springer-Verlag, 1985, 1–18.
61. B. Pernici and W. Vogel, "An Integrated Approach to OIS Development," *ESPRIT Technical Week '86*, Brussels, September 1986.

62. C. Rolland and C. Richard, "The Remora Methodology for Information Systems Design and Management," in *Information Systems Design Methodology: A Comparative Review*, (T.W. Olle, H.G. Sol, and A.A. Verrjin-Stuart, eds.), North-Holland, Amsterdam, 1982, pp. 369-423.
63. T. De Marco, "Structured Analysis and System Specification," *Prentice Hall*, Englewood Cliffs, NJ, 1979.
64. D.T. Ross, "Structured Analysis (SA): A Language for Communicating Ideas," *IEEE TSE*, 3(1), 16–34 (January 1977).
65. M. Jackson, *System Development*, Prentice Hall, 1983.
66. B.R. Konsynski, L.C. Bracker, and W.E. Bracker, Jr., "A model for specification of office communications," *IEEE Trans. Commun.*, COM-30(1), 27–36 (January 1982).
67. G. J. Nutt and P.A, Ricci, "Quinault: An Office Modeling System," *IEEE Comp.* 41–57 (May 1981.
68. N.M. Delisle, D.E. Menicosy, and N.L. Kerth, "Tools for Supporting Structured Analysis," in *Automated Tools For Information Systems Design—IFIP Conference*, (H.J. Schneider, and A.I. Wasserman, eds.), North-Holland Publishing Company, 1982, pp. 11–20.
69. D. Teichroew and E.A. Hershey, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Trans. Software Eng.*, SE-3(1), 41–48 (January 1977).
70. *Stoneman - Ada Reference Manual*, Department of Defense, Washington, D.C., (February 1980).
71. J. McDermid and R. Knut, *Life Cycle Support in the ADA Environment*, Cambridge University Press, Cambridge, 1984.
72. P. Atzeni and E. Carboni, "INCOD (A System for Interactive Conceptual Design) Revisited After the Implementation of a Prototype," in *R.55*, Italian National Research Council, Rome (February 1983).
73. R. Dahl, "Data Representation in the RAMATIC Tool and its Implications", SYSLAB Report 27, SYSLAB-G, Systems Development Laboratories, Gothenberg, 1984.
74. D. Eriksson, "The Conceptual Representation in the RAMATIC Tool," SYSLAB WP 91, SYSLAB-G, Systems Development Laboratory, Gothenberg, 1984.
75. D. Eriksson and U. Sundin "A Knowledge-Based Modeling Support System," SYSLAB Report 31, SYSLAB-G, Systems Development Laboratory, Gothenberg, March 1985.
76. L.A. Johansson, "Use Aspects of RAMATIC," SYSLAB WP 87, SYSLAB-G, Systems Development Laboratory, Gothenberg, 1984.
77. D.H. Torbjar, "Use of Graphics Systems in RAMATIC," *SYSLAB WP 86, SYSLAB-G*, Systems Development Laboratory, Gothenberg, 1984.
78. S. Sluizer and P.M. Cashman, "XCP: An Experimental Tool for Supporting Office Procedures," *IEEE Proceedings of the First International Conference on Office Automation*, IEEE Computer Society Press, New York, 1984, pp. 73–80.

FEDERICO BARBIC
ROBERTO MAIOCCHI
BARBARA PERNICI

AUTOMATED PROGRAM GENERATORS

INTRODUCTION AND DEFINITIONS

Automated program generators are a class of computer software that creates computer code based on user-specified criteria. Simply put, they are computer programs that create application-based programs. Automated program generators generally perform this function by offering choices to the user. Based upon user response, the program then, generates the appropriate program code. The resulting code may work as a stand-alone program or, in some cases, may require the presence of the program generator or some other program to work properly. Today, the most popular automated program generators create application programs in either the BASIC or Assembler programming language. The generated programming code has all the characteristics of human-generated code and, thus, can be displayed and modified.

Like many classes of computer software, automated program generators are referred to by many different names. For instance, applications generators are automated program generators that create applications programs requiring the presence of the applications generator to work properly. Other terms include, but are not limited to, code generators, systems generators, or applications development systems. For the sake of this article, all these terms are synonymous with automated program generators, unless specifically identified otherwise.

Automated program generators are problem oriented rather that procedure oriented. This approach permits the user to focus on solutions to a problem instead of the programming commands that comprise the solution. In this way, program generators are considered nonprocedural.

Automated program generators are more suited to business needs than scientific needs because they create applications programs that manipulate data, rather than programs that manipulate numbers through extensive and complex formulae. Ultimately, these programs enable such applications to be produced more easily, cheaply, and quickly, all providing for quicker program development.

Automated program generators function similarly to report writers and to query language facilities. All three classes of computer software are oriented to speeding up program development and maintenance and are intended to involve the end users more in the program development process. In fact, many developers of automated program generators boast that the end user can use them without any programmer assistance. However, in reality, automated program generators generally are best used by an individual (*a*) having working experience with the programming language of the programming code created by the program generator, (*b*) having a solid understanding of computer processes, (*c*) knowing how the generated program should operate, (*d*) having knowledge of how to speed up computer operations or (*e*) having enough experience to know what makes for a well-designed,

usable applications program. Thus, program generators are best used by skilled professionals, whereas the applications program generated can be used by the end user.

HISTORY

Automated program generators originated in the late 1960s on mainframe computers. By that time, programmers recognized that the greatest programming efforts were focused on program maintenance/modification. Typically, programmers spent 80% of their time maintaining programs throughout a program's lifetime. Additionally, programmers realized that many new programs contained routines or components common or similar to those contained in existing programs. Thus, as a means of increasing productivity, data processing departments began to take advantage of tools such as automated program generators.

Another factor, that of the increase in end user involvement in program development and maintenance, fostered the increased use of automated program generators. End users recognized that programs could be developed faster and closer to their specifications if the end users were more actively involved. As microcomputers appeared, end users demanded even more independence from the programming staff simply because the programming staff could not keep up with demand. Thus, it was at this time that automated program generators truly came into their own.

One of the first popular automated program generators was "The Creator." Available in 1980, The Creator worked on TRS-80 Model I microcomputers. The program created programs in BASIC, the most popular programming language for microcomputers at that time. The Creator became one of the most popular microcomputer-based automated program generators through some innovative marketing approaches by its creator, Bruce Tonkin. Its complete source code appeared in *80 Micro*, a major publication at that time.

However, the most recognized program of this type was "The Last One," by D. J. 'AI' Systems Ltd. This applications program generator was designed to run on all major microcomputers at the time (i.e., CP/M-based, Radio Shack, Apple II, and Commodore microcomputers). Like The Creator, The Last One generated BASIC language programs. The distributors of The Last One developed one of the most extensive promotional campaigns for the product, which greatly contributed to the popularization of this class of computer software within the microcomputer industry.

Through the mid-1980s automated program generators have appeared for virtually every popular microcomputer on the market. Each product has generally taken the features of its predecessors--The Creator and The Last One—and, for the most part, expanded upon them. Some, like "Quic-N-Easi," were written in Assembler for faster processing. Others, like "Formula" and "COGEN," produced programs with programming languages other than BASIC. Still others, like "Quickcode" or "dUTIL," generated programs that must be used with specific applications programs (in this case, dBASE II).

In recent years, there have been few significant breakthroughs in the development of automated program generators. Perhaps the lack of improvement can best be attributed to the general depression of the computer software market, as well as the unfulfilled expectations experienced by users of these programs. They found that virtually all automated program generators require working knowledge of the particular programming language used to create the applications program. Moreover, many users acknowledge that the programs generated still require some modification that can only be performed by a human. Thus, even though a computer has performed a large portion of the programming task, there is a need for a programmer to complete the application.

AUTOMATED PROGRAM GENERATOR COMPONENTS

Automated program generators are best characterized by the components in the applications programs created by the automated program generator. Automated program generators produce applications programs that contain the following four common data processing components:

Data input
Data storage/access
Data retrieval/reporting
Data calculations

Data Input

Data input permits an applications program to collect information. It is an essential part of the program created by the automated program generator because it is the component that permits the user to interact with the applications program. Automated program generators generally provide for the development of programs that use prompted screen-based forms as the means for data input. In the creation of the applications program, good program generators offer enough flexibility to design the form on the screen without requiring the aid of a paper-based design. Thus, a user can use pointing devices, such as arrow keys or a mouse, to design the form and alter as desired.

Good automated program generators also permit a user to redesign a data input screen, without requiring the recreation of the other components of the applications program. Some users of automated program generators take advantage of only the data input component to create stand-alone data input programs.

Automated program generators vary in other data input features that may be created in the applications program. Some offer very elaborate facilities for validating data entered. Most offer editing capabilities for data modification.

Data Storage/Access

The applications program created by the automated program generator must have the facility to store and retrieve the information captured by data input. Applications programs handle storage of the information as files containing single, two-dimensional tables. Users can think of the columns as fields of information and the rows as each entry or record. This form of storage can handle a number of applications satisfactorily. Some automated program generators can develop applications programs, having forms of storage that emulate the sophisticated means of access available through data base management systems. However, as the form of storage increases in complexity, so does the design of the automated program generator and the applications

program generated. Additionally, complex storage techniques require the end user of the applications program to understand all the nuances of his/her application and the relationships among the various information elements.

Automated program generators are capable of producing applications programs with differing forms of access, the second subcomponent of file storage. In fact, an automated program generator may determine the appropriate form of access, depending upon the application, and create the appropriate programming code. Common forms of access include sequential access, random access, indexed sequential, and hashing (see ACCESS METHODS, VOL. 1, this encyclopedia for additional description). An important component to all automated program generators is that the program generator—not the user of the automated program generator or the end user of the applications program—determines the method of access.

Data Retrieval/Reporting

Next to data entry, data retrieval/reporting is one of the most powerful components of applications programs created by program generators. In general, program generators produce a retrieval program with specific retrieval and report characteristics instead of producing generalized programs in which the user can specify the characteristics on demand. Naturally, if the user of the generated applications program requires different data retrieval or report criteria, a new applications program can be generated. Thus, reports tending to be used on a repetitive basis are the most useful to be generated by an automated program generator; *ad hoc* requests generally are not. Given the frequent need to generate reports, the report component of an automated program generator is often separate from the rest of the program and can generate a separate applications program.

Data retrieval is the capability of identifying information through specified criteria. Automated program generators vary as to the complexity of the criteria. For instance, a retrieval program may only be generated as long as the user does not specify more than three choices for any particular field. Recent automated program generators offer users a complete realm of options for retrieval, including all Boolean and relational operators for both string and numeric information. Many also offer the options for generating conditional statements.

Automated program generators differ in how the user of the program generator sets up the retrieval logic. Less sophisticated automated program generators require the user to structure the logic in the form acceptable to the programming language of the generated program. Thus if the programming language were, say, BASIC which only recognized two character variables, the user would be restricted to describe the logic in those terms (i.e., using variables such as A1 or A$). More sophisticated generators permit the use of English-like words and phrases and will translate them accordingly.

The kinds and variety of report formats capable of being created differ among automated program generators. Most generators can create programs producing columnar-type formats with breaks for subtotals and totals. Others provide for programs producing free-form reports, merged reports, and, often, structured noncolumnar reports, such as those used for mailing lists.

Calculations

The applications programs created by automated program generators excel in manipulating non-numeric information. Storing city names or retrieving and reporting on a particular city is an easy task for generated applications programs. Creating programs that perform complex mathematical calculations is not as easy to develop with automated program generators. All program generators can create programs that perform simple calculations; only the more sophisticated generators can perform more advanced and complex operations, such as conditional statements and trigonometric functions.

Often, the automated program generator's ability to perform calculations is limited to the capabilities of the programming language of the generated applications program. Thus, if the applications program is generated in BASIC, a user can perform much more advanced and complex calculations than one generated in COBOL.

A second limitation to calculations is based on the structure of the information. Complex scientific calculations are simply not feasible to be described within the context of an automated program generator. However, many automated program generators can create applications programs containing dummy variables or calculated fields to simulate semicomplex calculations.

COMMON APPLICATIONS OF AUTOMATED PROGRAM GENERATORS

The applications programs created by automated program generators function very similarly to file management programs and, in a few cases, to data base management systems, particularly in microcomputer-based applications. Thus, applications suitable for file management programs are generally good candidates for automated program generators.

Some examples include telephone directories, inventories, appointment calendars, expense reports, and investment portfolios. It should be noted that every application is restricted to a two-dimensional data storage structure. This tends to prevent the creation of multiple applications from the same information. Thus, an automated program generator cannot create an application program(s) that handles the integrated requirements of accounts payable, accounts receivable, and general ledger.

There are other applications that are not particularly appropriate to automated program generators. They include applications requiring *ad hoc* requests and applications containing complex mathematical calculations. Moreover, the generated application program may have certain limitations restricting the usefulness of an automated program generator for certain applications. For instance, a generated applications program will have an upper limit on parameters, such as the number of records, the number of fields, and the length of the fields an application will have. Although the limit has been extended in recent versions of automated program generators, the parameters of file management systems generally continue to exceed those of automated program generators.

By their nature, applications programs created program generators are generic. This characteristic may limit the practicality of certain applications without human intervention to optimize or restructure the applications program. Few automated program generators can create applications programs that take advantage of specific machine hardware and machine architecture.

EVALUATION OF PROGRAM GENERATORS

Determining the appropriateness of an automated program generator for a particular application is an extremely difficult and complex process. Clearly, it is a task that requires an individual with a solid foundation in data processing, as well as some idea of intended applications. Evaluation will require analyzing the following elements as well:

Target user of application
Programming language of generated applications program
System requirements
Program logic capabilities
Modification capability of applications program

Target User

Although automated program generators are used most successfully by programmers or someone knowledgeable in programming, the generated applications program may be used by individuals with fewer skills and programming experience. Thus, a user of automated program generators must be familiar with the level of complexity of the programs generated. Are the applications programs, for instance, user friendly and easy enough to be used by individuals inexperienced with computers? Is the applications program capable of validating user input and providing for assistance in those cases where improper information has been entered?

Unfortunately, determining the level of complexity for the program generated is, at best, a subjective effort and may require extensive use of, and practice with, the automated program generator. In many cases, it will be necessary to modify an applications program to suit a particular end user's level of computer experience.

Programming Language of Generated Applications Program

It is generally helpful if the programming language of the generated applications program is one with which the end user or the user of the automated program generator is familiar. The more familiar with a language a user is, the more likely the program can be modified to fit the user's specific requirements.

It is also essential to know if the applications program can stand alone or if the program requires the automated program generator or some other program to work properly. Generally, non-stand-alone applications programs create nonstandard programming code.

System Requirements

Until the computer world standardizes, system requirements are an important component to automated program generators. Because automated program generators produce generic code, generated applications programs will work on specific machines and their equivalents. In fact, the only limitation may be the dialect of the programming language of the applications program, not the machine brand or manufacturer. Thus, an automated program generator written in Microsoft BASIC, which generates applications in that language may create applications programs that are capable of working on all microcomputers on which Microsoft BASIC works.

Given the generic orientation of automated program generators, performance of the applications program may be, at best, slow. If the user of the program generator is familiar with the programming language of the generated applications program, he/she can modify the program to increase its speed.

At present, automated program generators do not have any specialized hardware requirements, especially with regard to those programs designed for microcomputers. Usually, the requirements for an automated program generator and its generated applications program are the same and reflect the requirements one would experience with a file manager. A typical automated program generator works with modest primary storage (the earliest versions required only 48K), a mass storage device such as a disk drive or hard disk, and a printer. The application ultimately will dictate the specific hardware requirements.

Program Logic Capabilities

The relative ease or difficulty of creating an applications program is determined by the automated program generator's program logic. Some automated program generators require a user to describe "solutions" in terms of program commands. In these cases, the generator becomes less nonprocedural and generally more difficult to use. More sophisticated and easier-to-use automated program generators permit the user to describe logic in English-like form. In turn, the generator translates the logic in the appropriate form.

Modification Capability of Applications Programs

The steps necessary to modify or maintain an application is another important aspect on which to evaluate automated program generators. Less sophisticated automated program generators require the whole application to be regenerated for any modification. More sophisticated generators may require only part of the application to be regenerated. Those generators requiring only partial reruns can often generate stand-alone program segments, such as prompted screen-based data entry forms for other applications.

THE FUTURE OF AUTOMATED PROGRAM GENERATORS

Since the advent of microcomputers, acceptance and use of automated program generators has expanded beyond the boundaries of the data processing department. Many users with limited data processing experience have taken advantage of automated program generators to create appropriate applications programs without the assistance of a knowledgeable programmer. However, automated program generators have yet to live up to their promise of being completely end-user products. Notwithstanding, automated program generators continue to prove themselves capable of saving time in the creation and maintenance of certain applications. With proper knowledge and understanding of their limitations, automated program generators have increased productivity, in some cases, by up to 58%.

In the near future, automated program generators should continue to extend their capabilities to generate a greater breadth of applications

programs. This development can take off in many directions including, but not limited to, more sophisticated methods of data storage and access and the integration of multiple applications.

Over the long term, automated program generators will be coupled with expert systems to generate extremely specific applications programs. Additionally, because it is anticipated that microcomputers will continue to increase their power, program optimization, particularly as may be required with generically generated programs, will become less of an issue. Thus, more users should embrace automated program generators. Toward this end, program generators will truly become end-user tools, with less need to modify the generated application program.

BIBLIOGRAPHY

Cowart, Robert, "Time Saving Software: Streamlining your dBASE Code, Creating Instant dBASE Programs, Graphing your dBASE Data," *A+: The Independent Guide for Apple Computing*, 2(12), 81–85 (December 1984).

Frank, Werner L., "Application Generator or Data Management?" *Software News*, 5(12), 43–45 (December 1985).

Leavitt, Don, "Program Generators: Letting the Computer Do the Work," *PC Week*, 1(50), 53–58 (December 18, 1984).

Lobell, R. F., *Application Program Generators: A State of the Art Survey*, NCC Publications, Manchester, England, 1983.

Louden, Bob, "The Last One, a Program Generator from D. J. 'A1'," *Infoworld*, 4(2), 18–20 (January 18, 1982).

Martin, James, *Application Development without Programmers*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

Moskowitz, Robert, "Three Program Generators," *Pop. Comput.*, 2(5), 146–150 (March 1982).

Shipley, Chris, "Program Generators Gaining Flexibility, Speed," *PC Week*, 2(29), 119–120 (July 23, 1985).

Stewart, George, "Program Generators: They're Not as Easy to Use as Some Advertising Copy Suggests," *Pop. Comput.*, 1(11), 112–122 (September 1982).

Tannenbaum, Michael, "Program Generators," *Pop. Comput.*, 2(4), 40–46 (February 1983).

Tonkin, Bruce, "The Creator," *80 Micro*, (36), 74–96 (January 1983).

EBEN LEE KENT

AUTOMOBILE MANUFACTURING INTELLIGENCE

(see Artificial Intelligence for Automobile Manufacturing)

BASIC

INTRODUCTION

BASIC is an easy-to-use programming language available on nearly every microcomputer. It was created in 1964 at Dartmouth College, quickly became popular on time-sharing computers, and helped to propel the microcomputer revolution during the 1970s. By now, BASIC is probably the most popular computer language in the world. It is provided on most mainframe computer systems, nearly all minicomputer systems, and every commercial microcomputer system. In fact, it's difficult to buy a microcomputer without getting some version of BASIC included free.

BASIC was originally designed as a beginner's language, as its name indicates—BASIC is an acronym for "Beginner's All-Purpose Symbolic Instruction Code." However, it has grown into a language rich enough for common use in commercial, scientific, and educational computing. Because of its simplicity, BASIC is presently the overwhelming favorite among languages taught to high school and college students.

Yet, since the mid-1970s, BASIC has been vilified by computer scientists for its lack of structuring aids. More recently, BASIC's lack of control structures led the Educational Testing Service to reject BASIC as its language for the Advanced Test in Computer Science despite BASIC's wide popularity. The American National Standards (ANS) committee, a voluntary industry group charged with developing national standards for programming languages, took this criticism seriously. In 1987, it issued a standard for "ANS BASIC, which contains modern control structures, device-independent graphics, file handling, and much more. It appears likely that, if widely adopted, it will end the controversy over BASIC and computer education. Seeing that ANS BASIC appears to be the direction in which future BASICs will tend, it is discussed at length later in this article.

HISTORY

The BASIC language was designed by Professors John G. Kemeny and Thomas E. Kurtz of Dartmouth College as part of a larger experiment in making computers easy to use. The two professors had introduced computing at Dartmouth in the mid-1950s and were intrigued with the idea of making computers readily accessible to liberal arts students. Their first attempt, with an LGP-30 computer, was a success, but Kemeny and Kurtz were eager to try a larger experiment as the LGP-30 was a "batch" system and could not accommodate many students.

Early History at Dartmouth

In 1960 or 1961, Kurtz visited John McCarthy, the inventor of LISP, at MIT and came away convinced that Dartmouth should try to develop its own time-sharing system. Kemeny agreed but argued that existing computer languages, such as ALGOL and FORTRAN, were simply too hard for most people to learn. The new time-sharing system, then, would need a new language—BASIC—so students could learn to program.

Kemeny started work on the first BASIC compiler in the summer of 1963 while a small team of undergraduates began writing the operating system. In February 1964, the GE-225 computer was delivered to Dartmouth, and the first BASIC program ran on May 1, 1964 at about 4 A.M.

This earliest version of BASIC contained the following 14 statements: DATA, DEF, DIM, END, FOR, GOSUB, GOTO, IF, LET, NEXT, PRINT, READ, REM, and RETURN. Variable names consisted of a single letter, or one letter followed by a digit. Arrays could have one or two dimensions; their names were single letters. The first version provided nine built-in functions: Sin, Cos, Tan, Atn, Exp, Log, Sqr, Rnd, and Int. Users could define their own functions by a single-line DEF statement. User-defined function names were three letters long, beginning with the letters "FN" and ending with a third letter. (The restrictions on variable, array, and function names were adopted to simplify symbol table management, as symbol table techniques were not well understood in the early 1960s. These restrictions lasted well into the mid-1970s in most dialects of BASIC.) Curiously, the first version of BASIC did *not* have an INPUT statement. Though it was designed to be interactive, user input was not added to the language until the third edition (in 1966).

Because BASIC was developed in conjunction with one of the earliest time-sharing systems, it was the first major language explicitly designed for interactive use. Indeed, fragments of Dartmouth's early time-sharing environment have been preserved within most implementations of the BASIC language. For instance, BASIC's hallmark, line numbers, were actually part of the time-sharing system's editing scheme and used for ALGOL and FORTRAN as well as BASIC. (Within BASIC, of course, they doubled as targets of GOTO and GOSUB statements.) Similarly, the RUN and LIST commands were not a part of the BASIC language but were commands to Dartmouth's time-sharing system. When other implementors adopted the BASIC language, they brought along its interactive environment, line numbers, and commands.

Once the first version of BASIC was in place, Kemeny and Kurtz and their colleagues and students immediately set about refining and extending it. Eight new versions of BASIC appeared between 1965 and 1982; each new "edition" was marked by a new manual. In a nutshell, these editions were added: *Second edition* (Fall 1964): semicolon as a print separator, arrays start at zero, an early version of the MAT statements. *Third edition* (1966): INPUT and RESTORE statements, MAT statements, Sgn function. *Fourth edition* (1967): RANDOMIZE, Rnd with no dummy argument, ON-GOTO, TAB, multiple assignments with one LET, string variables and arrays, multiple-line functions. *Fifth edition* (1970): text and random access files; internal subprograms; end-of-line comments; recursive user-defined functions; built-in functions for time of day, date, Len, Str$, Val, and Ord. *Sixth edition* (1971): external subprograms with arguments, libraries, automatic overlays, string concatenation and substring extraction, PRINT USING, libraries to handle machine-independent graphics. *Structured BASIC* (1975): modern control structures, long variable names, graphics statements. *Seventh Edition* (1978): "groups" of related external routines sharing common, private data; structure exception handling; recursive functions and subprograms; general matrix arithmetic in LET statements. *Eighth Edition* (1982): ANS-compatible.

Early History outside Dartmouth

While BASIC was evolving at Dartmouth, it began to flourish elsewhere. Dartmouth's original GE-225 BASIC and operating system were adapted for the GE-265 computer and sold by General Electric. It was the first widely popular time-sharing system, installed in several hundred sites around the world. GE then adapted this BASIC for use with its MARK I and MARK II time-sharing services. Other early versions of BASIC were propagated by DEC, IBM, NCR, HP, Wang, Xerox, the Universities of Maryland and Californ (at Berkeley), and others. By 1970, there were over 20 different versions of BASIC available on mainframes and minicomputers.

Even at this early date, the various dialects of BASIC had begun to diverge. During the 1970s, BASIC spread to microcomputers and even to pocket calculators; implementors often severely edited the language to fit it into the small memories then available or extended the language in nonuniform ways to add new features. By the mid-1980s, several hundred versions of BASIC existed.

Microcomputer BASICs

The first version of BASIC for a microcomputer was called Tiny BASIC, created by Dennis Allison in 1974 for the Altair microcomputer. It let users write simple programs in a subset of the BASIC language and, as it occupied only 2Kb of the Altair's total memory of 4Kb, left 2Kb free for the user's program and data. Naturally, many features of BASIC were dropped in order to fit into 2K of memory; there were no strings, no floating-point numbers, no MAT package, no files, and so forth. However, Tiny BASIC was published in newsletters and magazines, quickly became popular, and was adapted and expanded by various programmers.

In late 1974, William Gates and Paul Allen, both then students at Harvard University, developed the first commercial BASIC for a microcomputer. Again, they chose to implement BASIC on the Altair computer with its 8080 processor and 4Kb of memory. Unlike Tiny BASIC, which was designed to run on several kinds of computers, this BASIC was tailored for the Altair, and it provided floating-point arithmetic. Like Tiny BASIC, however, this version of BASIC was an interpreter that scanned a program's text repeate while executing it. In a sense, microcomputer BASICs fell back to the BA of 8 years before; the Altair BASIC of 1975 was very similar to Dartmouth' 1967 BASIC. Gates and Allen formed a company shortly afterwards, called Microsoft Corporation, and their various versions of BASIC have since become the most common BASICs on microcomputers. In particular, their PC-BASIC, distributed free with the IBM PC computer, has become very popular.

In 1976, Gordon Eubanks and Gary Kildall at the Naval Postgraduate School developed what would be the other major BASIC for microcomputers. Their original version was called BASIC-E; unlike previous microcomputer BASICs, it translated source programs into an intermediate form, which it then interpreted. This was a significant advance over strict interpreters, as

programs written in BASIC-E could be "compiled" into this intermediate form and then sold; the source program need not be distributed. This version of BASIC later evolved into the popular CBASIC.

During the next decade, many new BASICs appeared. Some, such as Applesoft BASIC for the Apple II computer, were widely popular. Many others have vanished without a trace. By 1980, new developments in microcomputer BASICs had come to a halt, with various Microsoft BASICs emerging as the most common versions of the language. But in the middle 1980s, dissatisfaction with the older Microsoft BASICs was sufficiently widespread to start another spate of BASIC implementations. BetterBASIC, True BASIC, Professional BASIC, QuickBASIC, Turbo BASIC, and the (abandoned) Macintosh BASIC are examples of the most recent crop of BASICs.

ANS BASIC

During the 10 years from 1975 to 1985, microcomputer BASICs regained the same ground that time-sharing BASICs had covered during 1965 through 1975; Files, graphics, strings, and rudimentary control structures were added to most versions of the language. Unfortunately, they were added in different ways to different dialects. As a result, BASIC is among the least standardized of languages, and programs written in one version often require heavy rewriting to run under other implementations. The ANS committee began to address this problem in 1974, and by 1978 had issued its first standard for Minimal BASIC. This standard codifies the syntax and semantics of the language corresponding to 1965 Dartmouth BASIC and has been widely adopted for microcomputer BASICs.

After finishing the standard for Minimal BASIC, the committee again began to consider a standard for the full ANS BASIC language. By this time, it was clear that memory sizes were quickly increasing on microcomputers and that the language need not be scaled down to fit microcomputers. Dartmouth's BASIC (7th edition) was chosen as a starting point, but the syntax was thoroughly overhauled and the language improved in many ways. By late 1984, the design was essentially complete and it was formally approved in 1987.

ANS BASIC is by any measure a large language, and it remains to be seen whether it will prevail over existing dialects. It provides many benefits not commonly found in microcomputer BASICS, such as machine-independent graphics, structuring constructs, multiple-line functions and subroutines, and so forth. This language could not have been implemented in early 4K microcomputers but fits well in machines with 128Kb, or more, of available memory.

MINIMAL BASIC EXAMPLES

Figure 1 shows a very simple Minimal BASIC program. Anyone comfortable with high school algebra should be able to decipher this program. Lines 100 and 110 set the variables *a* and *b* to have values 2 and 3, respectively. Line 120 sets the variable *x* equal to the sum *a* + *b* (i.e., 5), and line 130 prints this result. The program ends at line 140.

Each BASIC statement begins with a "keyword," which identifies the statement. Thus, a LET statement assigns a new value to a variable, a PRINT statement prints results on the terminal or screen, and the END statement


```
100 LET a = 2
110 LET b = 3
120 LET x = a + b
130 PRINT x
140 END
```

FIGURE 1 A Minimal BASIC program.

marks the end of the program. In addition, each line is identified by a line number. Line numbers must be present on all lines, must be strictly increasing and are used as targets for GOTO and GOSUB statements and their variants.

A slight change in the program allows the user to type in values for *a* and *b* while the program is running. Figure 2 shows the same small program but with LET changed to INPUT in lines 100 and 110. When this program runs, BASIC prints a question mark when it executes line 100 and waits for the user to type a number on his or her keyboard. It then assigns this number to the variable *a*. Line 110 again makes BASIC print a question mark and wait for input and then assigns this second number to *b*. The remainder of the program works as in the previous example.

A slightly more sophisticated example, shown in Figure 3a, prints a table of the numbers between 1 and 10, with their square and cube roots. Each column is labeled.

The REM statements in lines 100 and 110 introduce "remarks" or comments into a program. These lines contain notes about how the program works; they're intended for people reading the program, and BASIC ignores them. The PRINT statement in line 120 prints three labels; the commas between the labels mean that they should be lined up in columns. (Most versions of BASIC use columns 15 or 16 characters wide.) Lines 130 through 150 form a loop that executes repeatedly. The "loop index variable" *i* is first set to 1. The loop is then executed once. At the NEXT statement, *i* is increased by 1. If the new value is less than or equal to the loop's endpoint, 10, BASIC jumps back to the FOR statement. Thus, line 140 is executed 10 times, with *i* ranging from 1 to 10. Each time it's executed, it prints the current value of *i* and its square and cube roots.

The square root is indicated by Sqr, a function built into BASIC. Cube roots are not built into BASIC, so they are written as *i*^(*1*/*3*) or *i* raised to the 1/3 power. Because this PRINT statement uses commas to separate the output items, they are neatly lined up in columns. Figure 3b shows the output from this program.

```
100 INPUT a
110 INPUT b
120 LET x = a + b
130 PRINT x
140 END
```

FIGURE 2 Add two numbers typed from keyboard.

```
100 REM  Print table of numbers and square and cube roots.
110 REM
120 PRINT "Number", "Square Root", "Cube Root"
130 FOR i = 1 TO 10
140       PRINT i, Sqr(i), i^(1/3)
150 NEXT i
160 END
```

FIGURE 3a Print table of numbers and roots.

Number	*Square Root*	*Cube Root*
1	1	1
2	1.41421	1.25992
3	1.73205	1.44225
4	2	1.5874
5	2.23607	1.70998
6	2.44949	1.81712
7	2.64575	1.91293
8	2.828843	2.
9	3	2.08008
10	3.16228	2.15433

FIGURE 3b Output from program in Figure 3.

Figure 4 shows a longer Minimal BASIC program, illustrating many features of the language in a form compatible with most BASIC dialects. It converts measurements from one unit to another (e.g., feet to inches). The program begins by reading data built into the program and printing the converted results on the user's screen or terminal. It then loops, asking the user for units to convert, and prints the results.

The whole program in Figure 4 will be described briefly, and each statement will then be discussed in more detail. Overall, lines 100 through 120 introduce the program and define a function, Fnc, which converts one unit to another. Lines 140 through 180 read internal data, containing several measurements and conversion factors, and print the converted results. Lines 200 through 280 form a loop that asks the user if he or she wishes to convert any additional measurements. If so, the program reads the units and conversion factors from the terminal or keyboard and prints the results. Lines 500 through 540 are a subroutine that calculates and prints a converted result, and lines 900 through 920 contain the internal data used within the program.

Lines 100, 110, 130, and so forth, are comments. Comments are notes added for the benefit of people reading a program and are ignored by BASIC. In Minimal BASIC, each comment is introduced by the REM (remark) keyword and occupies an entire line. Because Minimal BASIC does not allow blank lines within a program, REM statements are also used as "blank" lines setting off blocks of statements.


```
100 REM     Conversion Program  (Minimal BASIC).
110 REM
120 DEF  Fnc(f, c1, c2) = c1 * f + c2
130 REM
140 READ  n
150 FOR  i = 1  TO  n
160       READ  f, m1$, m2$, c1, c2
170       GOSUB 500
180 NEXT  i
190 REM
200 PRINT "Do you want to do another conversion";
210 INPUT a$
220 IF a$="n" then 999
230 IF a$="N" then 999
240 PRINT "Enter amount, original unit, result unit,"
250 PRINT "multiplying constant, adding constant"
260 INPUT f, m1$, m2$, c1, c2
270 GOSUB 500
280 GOTO 200
500 REM
510 REM  GOSUB routine: calculate and print result.
520 REM
530 PRINT f; m1$; " = "; Fnc(f, c1, c2); m2$
540 RETURN
900 REM
910 DATA 2
920 DATA 6, feet, inches, 12, 0
930 DATA 22, Centigrade, Fahrenheit, 1.8, 32
999 END
```

FIGURE 4 Minimal BASIC program to convert units.

Line 120 defines a function, Fnc, of three numeric arguments. It returns the computed value of the expression c1 * *f* + c2, where new values for c1, *f*, and c2 are substituted in every call of Fnc. (BASIC, like most computer languages, uses the asterisk [*] to stand for multiplication because the usual multiplication symbol isn't found on computer keyboards. It also substitutes a slash [/] for the division sign.)

Line 140 reads a numeric value from the internal data pool into variable *n*. This value is found at the start of the data pool in line 910.

Lines 150 through 180 form a loop that increments the value of *i* from 1 to *n* (adding 1 each time). Within the loop, line 160 reads values from the internal data pool, and line 170 uses a GOSUB statement to call the subroutine that calculates and prints the converted result.

Lines 200 through 280 form another loop that repeatedly requests if the user would like to perform another conversion. Lines 200 to 230 print an explanatory message on the terminal or screen, read an input reply from the keyboard, and check to see if the reply is either *n* or *N*. If so, control jumps to the END statement at line 999 and the program stops. Otherwise, line 260 reads five input items (amount, unit names, and conversion constants) from the keyboard, and line 270 calls the subroutine that calculates and prints the results. Line 280 then transfers control back to line 200 to repeat the loop.

Line 500 through 540 form a "subroutine" reached by GOSUB statements. A GOSUB statement transfers control to another line number but remembers the line that contained the GOSUB. When a RETURN statement is executed, control returns to the lines containing the GOSUB. This is the only sort of subroutine provided in Minimal BASIC. Because subroutines are not organized into coherent constructs, they may have multiple entry and exit points, and control may even accidentally "fall into" a subroutine from the statements before it.

Every Minimal BASIC program must end with an END statement, which must be the last statement in the program. When control reaches this statement, execution stops. (Many versions of BASIC do not require an END statement.)

Several other features of Minimal BASIC deserve mention. First, all variable names are only one or two characters long. Variables do not need to be declared and are "global" to the entire program. Control structures are built from simple tests and jumps; more complicated control structures are not defined in Minimal BASIC. Data can easily be read from an internal data pool or from the keyboard. Neatly formatted output is simply handled by the PRINT statement.

DIFFERENT DIALECTS

As mentioned above, BASIC has developed as a fantastic variety of dialects, rather than as a single, unified language. In essence, each implementor freely improvised a language based on Dartmouth BASIC. The result has been chaos. The resulting dialects are recognizably BASIC but differ so widely that it is nearly impossible to write programs that will execute correctly in all the major dialects.

Figures 5a through 5d show a smaller program highlighting the problems with BASIC dialects. The program opens a text file named "MYFILE" and reads strings from it; it prints the second through fourth characters of each string on the screen, and that is all. As these programs illustrate, even small programs suffer from the dialect confusions of various BASICs. File management, loops, string manipulation, and the comment characters differ widely between BASICs. Note that in many versions of the language, the end-of-file condition must be treated as an error. Also note that although several dialects of BASIC provide a function named Mid$ for extracting substrings of a string, they disagree on the semantics of the function: Some provide a starting character position and length; others provide starting and ending positions.

```
100 '     PC-BASIC.
110 '
120 OPEN "MYFILE" FOR INPUT AS #1
130 '
140 WHILE NOT Eof(1)
150      INPUT #1, a$
160      PRINT MID$(a$,2,3)
170 WEND
180 END
```

FIGURE 5a Program in PC-BASIC.


```
100 !   VAX-11 BASIC.
110 !
120 ON ERROR GOTO 500
130 !
140 OPEN "MYFILE" AS FILE #1, ACCESS INPUT
150 !
160 WHILE 1% = 1%
170      INPUT #1, a$
180      PRINT MID$(a$,2,4)
190 NEXT
500 !
510 END
```

FIGURE 5b Program in VAX-11 BASIC.

```
100 REM   Applesoft BASIC.
110 REM
120 ON ERR GOTO 500
130 REM
140 LET d$ = Chr$(4)
150 PRINT d$; "OPEN MYFILE"
160 PRINT d$; "READ MYFILE"
170 REM Loop
180      INPUT a$
190      PRINT MID$(a$,2,3)
200 GOTO 170
500 REM
510 PRINT d$; "CLOSE MYFILE"
999 END
```

FIGURE 5c Program in Applesoft BASIC.

```
!   True BASIC.
!
OPEN #1: NAME "MYFILE", ACCESS INPUT

DO WHILE MORE #1
   INPUT #1: a$
   PRINT a$[2:4]
LOOP

END
```

FIGURE 5d Program in True BASIC.

AN ANS BASIC PROGRAM

This section presents several programs written in the ANS Standard BASIC. The standard supports line numbers, but as these programs do not include any GOTO or GOSUB statements, line numbers have been eliminated.

The first example, Figure 6, recasts the program shown in Figure 4. This version, however, is written in ANS BASIC rather than Minimal BASIC.

Despite its strikingly different appearance, this program is a close cousin of the Minimal BASIC version. The LET, PRINT, INPUT, READ and DATA, FOR, NEXT, DEF, and END statements are unchanged. Only the control structures have been revised. ANS BASIC allows a DO-LOOP construct that executes the lines between DO and LOOP repeatedly. (Control exits the loop when the EXIT DO statement is executed.) The GOSUB routine has been changed to a named subroutine, Convert, and all GOSUBs that called it have been changed to CALL statements. Variable names have been left untouched, although ANS BASIC allows variables and functions to have names up to 31 characters long.

Figure 7 contains another ANS BASIC program, highlighting more of the modern control structures found in ANS BASIC. This program simulates dice rolls in a game of "craps" and prints the results on the screen.

```
REM  Conversion Program (ANS BASIC).

DEF Fnc(f,c1,c2) = c1 * f + c2

READ n
FOR i = 1  TO n
     READ f, m1$, m2$, c1, c2
     CALL Convert
NEXT i

DO
    PRINT "Do you want to do another conversion";
    INPUT a$
IF  a$="n" or a$="N" THEN EXIT DO
    PRINT "Enter amount, original unit, result unit,"
    PRINT "multiplying constant, adding constant"
    INPUT f, m1$, m2$, c1, c2
    CALL Convert
LOOP

SUB Convert
    PRINT f; m1$; " = "; Fnc(f,c1,c2); m2$
END SUB

DATA 2
DATA 6, feet, inches, 12, 0
DATA 22, Centigrade, Fahrenheit, 1.8, 32
END
```

FIGURE 6 ANS BASIC program to convert units.


```
!    Play game of "craps."
!
FOR game = 1 to 10
     CALL Roll
     SELECT CASE dice
          CASE 2, 3, 12
                PRINT "You lose."
          CASE 7, 11
                PRINT "You win."
          CASE ELSE
                LET point = dice
                DO
                     CALL Roll
                LOOP UNTIL dice = 7 OR dice = point
                IF dice = point THEN
                   PRINT "You win."
                ELSE
                   PRINT "You lose."
                END IF
     END SELECT
NEXT game

SUB Roll
     LET die1 = Int(6 * Rnd + 1)
     LET die2 = Int(6 * Rnd + 1)
     LET dice = die1 + die2
END SUB

END
```

FIGURE 7 ANS BASIC program to play "craps."

The SELECT CASE statement chooses a block of statements to execute, based on the value of the *dice* variable. If *dice* equals 2, 3, or 12, the first block is executed. If it equals 7 or 11, the second block is executed. Otherwise, the CASE ELSE block is executed. This CASE ELSE block contains a DO-LOOP that repeatedly executes one or more statements until some test proves true. (Here, the test is placed at the end of the loop, but exit tests may be placed anywhere in a loop.)

A named subroutine, Roll, is defined at the end of the program between the SUB and END SUB statements. This is an internal subroutine and so shares all its variables with the rest of the program; however, ANS BASIC also provides external subroutines, which have their own local variables. The Roll subroutine has no parameters, but ANS BASIC allows subroutines (like functions) to have one or more parameters that can be used anywhere inside the body of the subroutine.

PHILOSOPHY

From the beginning, the BASIC language was designed to be easy to learn and use. As Kurtz has written, he and Kemeny wanted an approach that would

be almost self-instructional. At first, the two professors gave 4 hours of lectures before allowing students to write programs; by now the lectures are condensed into two videotapes with a total running time of less than an hour and a half. Because Kemeny and Kurtz were designing their own language, they freely modified it to make it easier to explain or understand. With thousands of Dartmouth students as guinea pigs, the language was quickly freed of needless complexities.

BASIC is carefully designed to provide an easy introduction to computing. Complicated parts of the language are reserved for experts and are not necessary for novices. Thus, declarations of variables are not needed for simple numbers and strings; only when students learn about arrays do they need to learn about the DIM statement. The simplest kinds of PRINT statements give neat and legible output, and so forth.

This philosophy is most apparent when BASIC is compared with its predecessors, FORTRAN, ALGOL, or ALGOL's successor Pascal. The simplest FORTRAN program immediately involves problems with variable names (names beginning with *I* through *N* are integers, and the remainder are real numbers) and the complicated FORMAT statement. The simplest ALGOL program requires a student to learn about declarations, BEGIN-END blocks, and procedure calls (for output). Furthermore, both languages force students to learn the distinction between fixed and floating-point numbers as soon as their programs involve both floating-point calculations and a loop (which must be governed by an integer).

In later days, Pascal has inherited many of these complications. Even the shortest Pascal program requires a PROGRAM part, declarations of all variables, and a BEGIN-END block. Again, students quickly run up against the complexities of fixed point and floating point. Thus, students learning to program in Pascal must tackle many of the hardest concepts in their first programs, whereas BASIC allows a slower and simpler introduction. As illustration, Figure 8 presents BASIC, FORTRAN, and Pascal versions of a program that computes and prints the inverses of numbers 1 through 10.

Arithmetic

BASIC was designed to have only one numeric data type. This was a difficult decision in the early 1960s, as existing computers did not have floating-point arithmetic designed into the hardware; instead, floating-point calculations were interpreted by software. Critics complained that programs would run faster if BASIC let programmers use integer arithmetic explicitly. Kemeny and Kurtz refused to compromise BASIC's simplicity by introducing two kinds of numbers and predicted that floating-point hardware would soon be added to computers. Within 2 or 3 years it was added, and floating-point calculations then ran as fast as integer arithmetic.

At present, microcomputer BASICs usually provide at least two different kinds of arithmetic—floating point and integer—and many have several varieties of floating point. Numbers and variables are usually taken as floating point, unless the number or variable name is followed by a percent sign to indicate an integer quantity. Ironically, the "efficient" integer arithmetic is slower than floating point on some computers! As floating-point hardware becomes cheaper and more widely used, demand for special integer arithmetic will probably fall and BASIC may return to having a single numeric data type.


```
!     BASIC.
!
FOR i = 1 TO 10
     LET r = 1/i
     PRINT i, r
NEXT i
END
```

```
*      Fortran.
*
     DO 10 i = 1, 10
          r = 1.0 / i
          WRITE (0,20) i, r
10 CONTINUE
20 FORMAT (1H , I3, 10H              ,F7.5)
```

```
{ Pascal. }

PROGRAM Inverses (input,output);
   VAR i:  INTEGER;
       r:  REAL;
   BEGIN
      FOR i := 1 TO 10 DO
         BEGIN
            r := 1/i;
            Writeln(i:3, ' ':10, r:12)
         END
   END.
```

FIGURE 8 A simple program in BASIC, FORTRAN, and Pascal.

Historically, BASICs have usually provided six to eight digits of accuracy in numeric calculations. As scientific and commercial use of BASIC has grow double precision (14 to 19 digits) has become a common option. In particular, six digits of accuracy proved utterly insufficient for commercial use because it yields inaccurate results for sums running beyond $10,000. It now seems that 10 digits is the minimum desirable precision.

Arithmetic has generally been computed in binary, reflecting the underlying hardware, but decimal arithmetic has become increasingly popular in recent years, as decimal round-off errors are more intuitive than binary round-off errors. ANS BASIC, as a result, specifies decimal arithmetic. (However, most versions of BASIC still provide binary arithmetic because binary numbers are more compact than decimal, and binary calculations are generally much faster than decimal.)

Strings

From 1965 onward, the designers of BASIC realized that *most* computer problems required processing text. Therefore, convenient ways to read and write sequences of characters, combine them, take them apart, and so forth, were essential in a useful programming language. BASIC provides "strings" for manipulating text. In general, strings do not have numeric values. Thus, while someone's age is most naturally represented by a number, names are generally kept as strings. BASIC strings can contain any number of characters, from none at all (the "empty string") up to some implementation-defined limit. Strings can be stored in string variables and combined via operators and functions that append strings together, extract sections of the strings, convert strings to numbers and vice versa, and so forth. In BASIC, strings are as easy and natural to use as numbers.

BASIC was not the first language to offer string variables, and it does not even have the best string-handling facilities. SNOBOL, for instance, has far more sophisticated string functions than BASIC. Surprisingly, though, BASIC is the only major language to provide a flexible string data type. Many modern languages now provide fixed-length strings, or strings of flexible size that always use a fixed amount of storage, but BASIC continues to be the only common language that allows simple string manipulation. Early microcomputer BASICs generally had a rather small limit on a string's maximum size (255 characters was common) and some even required strings to have a constant length, but most BASICs have generally allowed longer strings. By now, strings of 32,767 characters are common, and some versions provide even longer strings.

Manifest Types

Because BASIC's inventors wished to avoid the nuisance of declaring variables, they made the type of each variable *manifest*. That is, by inspecting a BASIC variable you can tell its type: a1 is a numeric variable, a1$ is a string variable. Matrices were instantly identifiable because they appeared with parentheses and subscripts or in MAT statements. In early versions of the language, names of variables could be only one or two characters long. Function names were three letters long, and user-defined functions began with "FN": FNA, FNB, and so forth. These rules, taken together, guaranteed that any type of variable or expression could be recognized without inspecting any context outside the statement itself.

Making types manifest has two important consequences. First, programmers can read and understand a block of statements without having to look at the surrounding context. This is not possible in languages such as FORTRAN, PL/I, or Pascal, where the meaning of a statement such as $a := b/2$ depends on the declarations of a and b. Second, compilers or interpreters for BASIC have an easy task. Not only is "look ahead" avoided, but even symbol table management is easy.

English Syntax

Most BASIC syntax consists of simple English words arranged as in a sentence. Because beginners often have difficulty with the concept of assignment, confusing it with an assertion of equality, BASIC's assignment statement tells the beginner how to read aloud an assignment statement: "LET A = B + C."

Every statement begins with a keyword, making it easy for both programmers and compilers or interpreters to decipher the statement.

Punctuation is kept to a minimum and is seldom used in simple BASIC programs; only the PRINT and PLOT statements require students to find the comma and semicolon on the keyboard.

Machine Independence

Dartmouth BASIC was designed as a "high-level language" that hid the details of the underlying computer from its users. Users never needed to know exactly how numeric expressions were calculated, how files were saved on disk, how string storage was allocated and freed, and so forth. This made BASIC easier to use and also allowed Dartmouth to switch computers several times without users noticing any difference. When graphics support was added to Dartmouth BASIC in the late 1960s, it was added in a machine-independent way. Many different kinds of plotters and CRTs were used from BASIC, but the same program would work on any of them—only one line, stating the plotter type, needed to be changed.

This is a point hardly worth making except to note that microcomputer BASICs have often headed the other way. In particular, graphics have almost always been implemented in a very machine-specific way, making it difficult to transfer graphics programs from one brand of computer to another. This is directly contrary to BASIC's philosophy and will probably be amended in future versions of the language. (In particular, ANS BASIC defines graphics in a completely machine-independent way.)

IMPLEMENTATION STRATEGIES

BASIC has been implemented in a remarkable variety of ways, including compilers and several forms of interpreters. Three general strategies have been adopted for interpreted versions of BASIC: *textual* interpretation, *semicompiled* interpretation, and *incremental compiler* interpretation. In addition, strict compilers for BASIC also exist though they are rarer than interpreters.

Textual interpretation, in essence, scans the characters of the program repeatedly in order to execute it. This approach is easy to implement but has severe drawbacks. For example, comments must be scanned and discarded each time they are encountered; thus, they slow down a program's execution speed. Line numbers must also be scanned repeatedly, and variables must be scanned and looked up within symbol tables for every use. Because this technique is so slow, many textual interpreters keep programs in a condensed form internally: Keywords are replaced by single characters, line numbers are stored in internal formats, and so forth. At present, various versions of Microsoft BASIC are the leading textual interpreters.

Semicompiled interpreters are somewhat rare for BASIC, though they are common for Pascal and Modula. Here, the language system consists of two parts: a compiler, which translates the textual program into a more compact series of intermediate codes (often resembling machine instructions), and an interpreter, which executes these intermediate codes. When the user gives a RUN command, the compiler translates the entire program into the intermediate code and then begins to execute it. Outstanding examples of this strategy are the CBASIC system and True BASIC.

A third strategy has been the incremental compiler. Here again, the language system consists of a compiler and interpreter. However, incremental compliers scan each line as the user enters it and (if syntactically correct) reduce it to an intermediate form. The expectation is that compiling the entire program when a user types RUN will take several seconds or longer, but the delays after entering each line are hardly noticeable. This approach guarantees that users cannot enter lines containing syntax errors (such as "LEET A=1") though it cannot rule out lines containing run-time errors (such as "LET A=1/0"). Incremental compilers are not well suited for complex languages, however, as they may have difficulty in discovering the types of nonmanifest names. (Thus, ANS BASIC gives trouble with functions that can look like variable names, subroutine parameters that may be arrays, and so forth.) Incremental compilers for BASIC have included DEC's PDP-11 RSTS BASIC system and BetterBASIC for the IBM PC.

A final strategy for BASIC has been compilation to machine instructions. Dartmouth's own versions of BASIC have always been compiled, and, indeed, BASIC is an easy language to compile. Several companies have written compilers for microcomputer dialects of BASIC because many business applications written in BASIC perform too slowly when run under textual interpreters. It is interesting to note that because BASIC floating point, strings, files, and graphics are not built into the microcomputers, these aspects of BASIC must always be interpreted. Therefore, compiled BASICs should outperform interpreted BASICs only in integer arithmetic, GOTOs, and GOSUBs. (In practice, however, complied BASICs generally do provide faster floating point, strings, graphics, and so forth, than interpreted BASICs because the routines providing these functions are better written than the interpreter's routines.) The best-known compliers for BASIC on microcomputers are the Microsoft BASIC Complier, QuickBASIC, and Turbo BASIC.

UNUSUAL FEATURES OF BASIC

Though BASIC is most directly comparable to Pascal, FORTRAN, and so forth, it also differs remarkably from these languages. Thus, Pascal is a classically "pure" algorithmic language, but most variants of BASIC contain sublanguages for matrix arithmetic, internal data, and string manipulation, and microcomputer versions commonly contain statements to manage graphics, sounds, device I/O, and so forth.

A few of BASIC's innovations are still not widely available in other languages and, perhaps, deserve some special note. Though some are small details, they contribute to BASIC's ease of use.

Immediate Mode

Because BASIC is so widely interpreted, some sort of immediate mode is almost always provided for BASIC. That is, statements can be typed in place of commands and take effect immediately. Thus, BASIC can be used as a simple calculator; more importantly, users can inspect malfunctioning programs and debug them by viewing and changing the values of variables. This feature is so valuable that virtually no BASIC is provided without it (aside from pure compilers).

Easy Input and Output

Reading numbers and strings from the keyboard is very easy in BASIC, and displaying them on the screen is just as simple. This is now becoming true in other languages, such as FORTRAN, but BASIC was unique in this regard for a decade. Numbers can be entered in most common forms and are automatically printed in the simplest format. Thus, users may enter the number 1 in any of the following ways: 1, 1.0, and +1.0e1. Contrast this with Pascal, a relatively modern language, where the number .25 is disallowed because real numbers must have at least one digit before the decimal point. Also, assignments such as *a* := 1e5 are disallowed for an integer *a* because 1e5 has the form of a real number, although in fact its value is an integer. BASIC also automatically repeats a request for input from the keyboard if the user has made a mistake in entering replies, rather than halting the program with an error message.

On output, the number 1 is, by default, printed simply as 1. Numbers with decimal parts are generally printed with a decimal point and the trailing digits: 1.23. Numbers larger or smaller than a certain range are automatically printed in scientific notation: 1.3e + 20. Thus, users always see numbers in a natural form. More detailed control of output format is available for advanced users, by a PRINT USING statement, but novice users need never see formatting control. Even then, the PRINT USING statement has a fairly simple way to describe the way the result should look: In essence, the user presents both the number and a "picture" of how the result should be formatted.

Graphics and Sound

The earliest known use of graphics from BASIC dates from the late 1960s when Professor Arthur Luehrmann at Dartmouth College devised a way to attach plotters to teletype terminals. He designed a collection of subprograms that let students plot, instead of print, their output. These subprograms worked with any kind of plotter; the students needed to change only one line in their program, identifying the plotter name, to have the same program work on different kinds of plotters. Thus, BASIC's first graphics were "device independent" because they let users write programs without having to include any instructions that would not work on every kind of plotter. Later versions of Dartmouth BASIC simplified plotting by adding special statements to the language, rather than relying on subroutine calls. Again, the same program worked on every output device.

Unfortunately, graphics were reinvented for microcomputers in the mid-1970s, and the microcomputer graphics were *not* device independent. Typically, they required users to specify all plotting in terms of "pixels" (the glowing phosphor dots on the computer's screen). Because every brand of computer had different numbers of pixels on its screen, programs were not readily portable between brands of computers. Furthermore, colors were generally indicated by number rather than by name, making it difficult for programmers to remember how to draw in red, for instance, and contributing to the difficulty in transporting graphics programs from computer to computer.

On the other hand, microcomputer graphics quickly evolved toward a "bit-mapped" style, which allowed quite different operations than earlier plotter output. Screen images could be stored into variables and later used to replace or alter portions of the screen. "Area filling" and "flooding" let users manipulate areas on the screens and not simply draw lines.

ANS BASIC restores the early device independence of graphics for BASIC. Based on the international Graphics Kernel Standard (GKS), it allows sophisticated graphics manipulation based on primitive operations carefully designed to work on many different brands of computers. As in Dartmouth BASIC, all graphics are described in "user coordinates," that is, the terms most natural for the problem. BASIC converts these coordinates to actual pixel locations on the screen, thus allowing the same program to run unchanged on computers with more or fewer pixels on their displays. A wide repertory of statements allows the programmer to find how many different colors can be displayed on the screen, set colors, and so forth. Because these developments make programming graphics output easier and also remove needless machine dependencies, we can expect them to be adopted quickly in most versions of BASIC.

Microcomputer BASICs also introduced "sounds." Because many microcomputers contain sound generators, it seems natural that BASIC users be given some way to produce and control sounds. Early versions of BASIC provided very low-level control over sounds—generally describing sounds in terms of frequencies and machine clock cycles—but more advanced BASICs have provided ways of describing sounds in less machine-dependent ways. The most advanced even allow users to describe melodies in terms resembling western musical notation rather than requiring each note be programmed as a pitch and frequency. As sound generation hardware matures, we can expect further extensions to BASIC allowing, for instance, multivoice melodies and speech output.

Matrices

Since 1965, BASIC has contained a modest "matrix package" of statements, allowing simple matrix arithmetic. Matrices may be assigned constant values, identity matrices, or other matrices; they may be added, subtracted, multiplied, inverted, and multiplied by scalars; and they may be read from or written to files. More specialized functions allow dot products, determinants, and the like. Though these statements do not add computational power to the language—as users could write the matrix algorithms for themselves in BASIC—they make BASIC a handy language for solving systems of linear equations, working with Markov chains, and so forth. Some BASICs perform matrix operations with greater precision than other arithmetic operations, as this helps reduce round-off error in matrix calculations.

Like FORTRAN but unlike Pascal, subroutines can accept arrays of different sizes as arguments. This makes it possible, for instance, to write one subroutine that sorts an array of any size. (Unlike FORTRAN, subscripts are checked to make sure they are in bounds.) In some versions of BASIC, arrays can be made larger or smaller as the program runs.

Data

The READ and DATA statements appear to be unique to BASIC. This is curious, as they fill a common need. Many programs require a certain amount of built-in data that are used to initialize some variables. For example, a compiler might contain the text and precedence number of each keyword and operator in the language it compiles. In a language such as Pascal, this data must be kept in a separate file or individually assigned to variables via assignment statements. In BASIC, however, this can be written more tersely as one READ or MAT READ statement followed by a DATA statement listing all the initial values.

Random Numbers

BASIC also appears to be the only popular language to have a built-in random number generator. Random numbers are, of course, commonly used in game programs but are also useful for other, more serious applications, such as educational software, CAI development, statistical analysis, and so forth. As good random number generators are difficult to write, Rnd has been a standard part of BASIC since the beginning.

MISSING FEATURES

Most versions of BASIC still lack adequate structuring for programs, and this is (at present) BASIC's greatest need. Dartmouth BASIC and ANS BASIC both define the full complement of modern control structures, including loops, calls, if-then-else statements, case statements, and so forth, and we can expect these structures to gradually become commonplace in BASIC. Even ANS BASIC, however, does not supply the answers to all complaints about BASIC.

Records

For most simple programs, variables and arrays, along with procedure calls, form an adequate way of structuring data. Many users have found, however, that they would like some facility, such as Pascal's records for more complicated data structures.

Modules

ANS BASIC does not support externally compiled subroutines, but extensions to ANS, such as True BASIC, do. A logical next step is to allow users to group routines together that work on shared data—sharing the data among these routines but hiding it from all other routines. Such a way of grouping procedures into modules has been provided by Modula and Ada, and even by Dartmouth BASIC and True BASIC, and will probably be introduced into other commercial versions of BASIC in the near future.

More Devices

As microcomputer manufacturers continue to develop new devices and attach them to microcomputers, there will be continued pressure to adapt BASIC to handle these devices. It is pure speculation as to which devices will be supported in the future, but one can imagine BASIC statements to drive speech synthesizers, direct graphics to a printer (rather than the screen), handle more specialized graphics hardware, such as sprites and bit planes, and so forth.

BIBLIOGRAPHY

BASIC is a language more widely used than studied; thus, theoretical works on BASIC are quite rare, whereas introductions, handbooks, and reference manuals flourish. Today, there are hundreds of books available that describe the BASIC language in one form or another. Each book is, or course, more or less tailored to one dialect of the language.

Introductory Books

Kemeny, J. G., and T. E. Kurtz, *Structured BASIC Programming,* John Wiley and Sons, New York 1987. (Introduction to programming using the True BASIC language.)

Kemeny, J. G., and T. E. Kurtz, *BASIC Programming,* 3d ed., John Wiley and Sons, New York 1980. (Introduction to programming using a traditional Dartmouth BASIC.)

Dwyer, T., and M. Critchfield, *BASIC and the Personal Computer,* Addison-Wesley, Reading, MA, 1978. (Introduction to programming using traditional BASIC.)

Morrill, H., *Mini and Micro BASIC*, Little, Brown, Boston, MA, 1983. (Introduction to programming, using PDP-11, Apple II, and TRS-80 BASICs.)

Nevison, J., *Little Book of BASIC Style,* Addison-Wesley, Reading MA, 1978. (Excellent guide to writing clear programs in traditional BASIC.)

Luehrmann, A., and H. Peckham, *Hands-On BASIC for the IBM PCjr,* McGraw-Hill, New York, 1983. (Introduction to programming, using the PCjr BASIC.)

Standards and Manuals

ANSI, *American National Standard for the Programming Language Minimal BASIC,* X3.60-1978, American National Standards Institute, New York, 1978. (The standard definition of Minimal BASIC.)

ANSI, *American National Standard for the Programming Language Full BASIC*, X3.113-1987, American National Standards Institute, New York, 1984. (The source for what this article calls ANS BASIC.)

IBM Corporation, *BASIC*, IBM Corporation, 1982. (Personal Computer BASIC interpreter. The most common version of BASIC on microcomputers. Produced for IBM by Microsoft Corporation. Similar to most Microsoft BASICs on Intel 8088 microprocessor computers.)

Summit Software, *Better*BASIC, Summit Software Technology, Inc., Wellsley, MA, 1984. (IBM PC BASIC extended by adding separate compilation, types, etc.).

Dartmouth BASIC, Kiewit Computation Center, Dartmouth College, Hanover, NH, 1970,1975,1978,1982. (Manuals for Dartmouth BASICs, editions 5–8.)

Elliott, B., *True BASIC Reference Manual,* True BASIC Inc., Hanover, NH, 1985. (Kemeny and Kurtz's new version of BASIC for microcomputers; based on the ANS Standard for BASIC.)

Digital Equipment Corporation, *BASIC on VAX/VMS Systems,* Order No. AA-L336A-TK, Digital Equipment Corporation, Maynard, MA, 1982. (A very popular BASIC for the VAX minicomputer.)

Douros, L., *The BASIC/VM Programmer's Guide,* Prime Computer, Inc., FDR3058-101A, Framingham, MA, 1980. (Typical of many BASICs found on minicomputers.)

Historical Books and Articles

Kemeny, J. G., and T. E. Kurtz, *Back to BASIC,* Addison-Wesley, Reading, MA, 1985. (BASIC's inventors tell its history, guiding philosophy, what went wrong, and what the future may bring.)

Kurtz, T. E., "BASIC", in *History of Programming Languages,* (R. L. Wexelblat, ed.), Academic Press, New York, 1981, (One of BASIC's inventors gives a detailed history of BASIC at Dartmouth College.)

Kemeny, J. G., and T. E. Kurtz, "Dartmouth time sharing," *Science*, 162, 223–228. (1968). (An overview of Dartmouth College's time-sharing system, the source of the NEW and SAVE commands, etc.)

Dijkstra, E. W., "Go to statement considered harmful," *Commun ACM,* 11, 147–148 (*March 1968*). (The opening blast in the great structured-programming controversy.)

BRIG ELLIOTT

BENCHMARKING PROGRAMMING LANGUAGES

INTRODUCTION

A benchmark was originally a mark on some permanent object indicating elevation. The mark served as a point of reference from which measurements could be made in topological surveys and tidal observations. Contemporary usage indicates an object that serves as a standard by which others can be measured. In particular, benchmarks for computers are generally a set of performance measurements that allow the comparison of different hardware/software combinations.

How should one proceed when comparing programming languages? Obviously, an empirical approach would be preferable. Benchmark programs provide such a rigorous approach to evaluating languages. A benchmark program is a routine (program) written in different languages to perform a standard task. Different routines can be written to test the different dimensions of a language. Each of them will uncover particular strengths and weaknesses of a language when executing a particular task. Benchmarking thus allows broad comparisons to be made among the different programming languages. But the comparisons will not give a conclusive "faster language". Benchmarks only show the capabilities of a language at performing specific tasks. A person may then select a language that would fill their needs.

As an example, the iteration speed of a language is one particular dimension of a language that can be measured. This benchmark is usually implemented as an empty loop. The computer does not do anything in the loop 10,000 times. Some examples of these tests have shown that on the IBM PC, BASICA takes 5 seconds, whereas COBOL takes 3.3 seconds.

A benchmark to test integer addition could also be used. This test might consist of incrementing a variable by one until it equals 32,767, the limit for a 2-byte signed integer. Such a test proves useful in evaluating time-critical operations.

For mathematical and scientific operations, a third benchmark, floating-point arithmetic, is more critical. Because double-precision numbers are too large to fit into the 8088's registers (IBM PC and compatibles), this test frequently causes the microcomputer to pass the mathematical activity into memory.

In some applications, string handling is important. A fourth benchmark could be string concatenation. This test would assign strings to two variables and then combine them into a third. Additionally, the test could involve finding substrings within strings.

A test that frequently produces the greatest variation between languages is the table lookup. The primary reason is that languages vary drastically in the ways in which they store data in tabular format internally.

Finally, consider a test in which language performance at handling files and the relative efficiency of the interaction with the operating system are important. Such a benchmark tests the speed with which a language handles disk I/O and is an example of a routine that goes outside internal memory to disk.

Possibly, the most frequently quoted benchmark program is the Sieve of Eratosthenes [*1*]—a simple procedure developed in the third century B.C. to find prime numbers. Frequently called a "number crunching" program, it actually involves very little arithmetic and no floating-point calculations at all. In the classic sieve procedure, the natural numbers are arranged in order, every second number after 2 is crossed out, every third number after 3, and so on, crossing out every nth number after n. The numbers that are not crossed out, which "pass through the sieve," are prime numbers.

One characteristic of this benchmark program is that it avoids multiplication and division because microcomputers that do not have native instructions for these operations usually perform them slowly. In fact, because this program does a lot of looping and array subscripting, one may conclude that it is biased toward machine-code compilers.

Other popular benchmarking programs are "Puzzle" [*1*], "Ackerman's Function" [*1*] and the "EDN Benchmarks" [*2*].

BENCHMARKING METHODOLOGY

Programming languages may be classified as low level or high level, general purpose or special purpose, procedural or nonprocedural, and compiled or interpreted. Each of these characteristics of a language affects some aspect of its performance.

Low-level languages give programmers direct control over details of computer hardware, such as memory locations, microprocessor registers, and I/O ports. High-level languages distance programmers from these details, forcing them instead to work with abstractions such as files, arrays, and variables. The lowest level is machine language, and the highest level (at least in theory) is a "natural language" like English.

Most languages are created to serve a specific purpose. However, many of these languages are extraordinarily flexible, and their ultimate uses far exceed the concrete plans of their designers. Special-purpose languages, on the other hand, enable programmers to solve narrowly defined problems or unusual applications, such as systems programming, where one must be able to control the details of the hardware.

Procedural programming languages require the user to specify a set of operations to be performed in a specific sequence. Unlike native machine language and assembly languages that require instructions in a form very special to a particular microprocessor, procedural languages are designed to be independent of particular machines. Although a procedural language specifies how something is to be accomplished, a problem-oriented language specifies what is to be accomplished. The closer the programmer can come to stating the problem without specifying the steps that must be taken to solve it, the more nonprocedural the language is.

On the most basic level of hardware, a computer can execute only those instructions written in its native machine code. This means that a program written in a high-level language must be translated into machine code before it will run. Two types of language translators can do this: compilers and interpreters. A compiler first translates the whole program into machine language, and the machine language version is then executed by the computer. An interpreter translates a given statement and then executes this

translation (machine language version) of the statement. Thus, the interpreter must repeat the translation of a given statement every time it is to be executed—a rather inefficient process. (The times for interpreted BASIC versus compiled BASIC in Table 1 indicate the extent of this inefficiency.)

Experimental Design

From a more global perspective, the solution of a problem actually involves the design, coding, and testing of a program, that is, the entire development process. Different aspects of the language selected affect the different phases of development.

In a labor-intensive environment like software development, the total program development time actually becomes much more critical than execution speed. Studies by IBM have shown the correct selection of a language can improve development times by as much as a factor of 10. Thus, an interpreted language like APL, although slower than compiler languages, may be a better selection if the dominant criterion is total time to project completion.

Most people, however, focus on measures directly related to performance, such as speed or memory requirements. In fact, there is an unfortunate tendency to look only at the final number—the execution time. This tendency has been nicely paraphrased in the third of the "not-so-golden rules of benchmarking" [3]: "Conditions, cautions, relevant discussion, and even actual code never make it to the bottom line when results are summarized."

Even the performance measures themselves can be contradictory and misleading. No one measure can predict how fast a computer will do hundreds of different jobs under various conditions. The one cited most often is MIPS, or how many million- instructions- per second a computer executes. Some vendors tout that benchmark; critics say MIPS stands for a "meaningless indicator of processor speed" because MIPS measures only how fast a machine's insides churn, not how fast its work appears on a screen.

There are no consensus guidelines on benchmarks and how vendors come up with them. Comparative claims of hardware vendors are often not based on any real head-to-head testing at all [4].

Similarly, Donald Knuth argues that we have comparatively little information about programming languages [5]. "We think we know but our notions are rarely based on a representative sample of the programs which are actually being run on computers."

Thus, in an empirical study to benchmark programming languages, one should focus on determining a representative sample for the experimental design. The most important consideration, however, is the selection of criteria. For example, a recent study [6], examines as possible criteria iteration speed of a language, integer addition, floating-point arithmetic, string handling, and speed of disk I/O. For these six criteria, the study develops six benchmark programs in 10 different languages, thus allowing for broad comparisons among the languages. One could then use the results of the study to determine ways in which a language's capabilities might mesh with one's needs, *if* the criteria were appropriate. To emphasize this last point, if one's criteria were in fact "ease of use" or "time to learn," the study would be useless.

The final consideration in the experimental design should be to introduce as little variation as possible into the results. Tetewsky, in his study of FORTRAN compilers [7], gives examples of this problem. Specifically, when one compares a compiler that runs on the IBM PC with one that runs

TABLE 1 Solution Times in Seconds

Benchmarking Results

Algorithm C2

Problem	BASIC Interp.	BASIC Compiler	FORTRAN IBM v. 2.0	FORTRAN Prof.	C-86	MWC 1 rv	MWC 2 rv
rand1	21.200	0.400	0.120	0.088	0.110	0.076	0.076
rand2	50.800	1.600	0.318	0.252	0.264	0.210	0.176
rand3	87.200	3.000	0.560	0.440	0.472	0.374	0.308
rand4	114.200	4.200	0.758	0.594	0.638	0.482	0.406
rand5	123.800	4.600	0.824	0.648	0.692	0.526	0.440
rand6	24.200	0.600	0.208	0.166	0.186	0.144	0.132
rand7	54.800	1.600	0.330	0.274	0.286	0.230	0.186
rand8	89.600	3.200	0.572	0.452	0.484	0.384	0.308
rand9	100.600	3.800	0.648	0.516	0.560	0.428	0.362
rand10	146.400	5.600	0.968	0.770	0.812	0.638	0.516
rand11	14.800	0.400	0.088	0.066	0.076	0.056	0.056
rand12	22.000	0.800	0.144	0.110	0.120	0.088	0.078
rand13	48.000	1.800	0.306	0.242	0.264	0.210	0.166
rand14	67.600	2.600	0.452	0.352	0.386	0.296	0.242
rand15	105.600	2.000	0.584	0.450	0.506	0.394	0.374
Total:	1070.800	36.200	6.880	5.420	5.856	4.536	3.826
Average	71.387	2.413	0.459	0.361	0.390	0.302	0.255
Ratio/	155.640	5.262	1.000	0.788	0.851	0.659	0.556

on an Otrona Attache, one is comparing more than the compilers—one is also comparing the computers.

Additionally, to be fair in comparisons of compilers, for example, one should attempt to set as many variables as possible in the same way from compiler to compiler. For instance, compilers have a variety of switch settings that disable or enable options such as optimization, subscript checking, and default storage.

Hardware Performance Considerations

The typical microcomputer has a small word size, low central memory, fixed-point arithmetic, and slow processing times. The types of programs used to test performance in the sample experiment to follow have low data requirements and the computer codes are often short. Thus, fixed-point calculations and processing time are more crucial for the type of programs analyzed here.

Arithmetic in the PC

There are three different types of numbers that are important to computer users: integers, fixed-point numbers, and floating-point numbers. Integers are whole numbers with no fractional part, and because they have no fractional part, they are infinitely precise. When performing arithmetic with integers, the results are integers. Addition, subtraction, and multiplication of integers always give a result that is an integer. Integer division will return a result that is an integer with a remainder where it exists.

Fixed-point numbers are numbers that can be a whole part and a fractional part. In general, any number that has a fractional part and an integer part is considered to be a fixed-part number. This type of number is not always exactly precise.

Floating-point numbers differ from fixed-point numbers in that they allow for a large range of numbers at the sacrifice of precision. The need for a larger range of numbers becomes clear when describing very small or very large quantities. With the floating-point system, eight digits of precision in the mantissa and two digits for the exponent have been allowed. Including the decimal and spaces, it takes 16 columns to express any number from 0.00000001 E−99 through 99.999999 E+99. To express the same range of numbers in fixed point would require 208 columns (including sign and decimal point). This also results in a loss of precision.

Most practical situations either require exact precision or a reasonable approximation. Because of this, computers usually manipulate only two of the three types of numbers: integers and floating point.

A Sample Experiment

To examine specific issues in detail, let us design and conduct a sample experiment. The general purpose of this experiment is to explore and initiate discussions on the "ideal" programming language for implementing optimization models on microcomputers.

The microcomputer explosion has brought about the development of many powerful programming languages. Some of the languages are similar to the ones used on large computers. Others are special-purpose languages on large systems that are used as general-purpose languages on microcomputers because of their faster execution. Some have been specially adapted for micro-

computer programming. Small-scale applications of mathematical programming algorithms could very well be implemented at less cost on a microcomputer than on a large- or medium-size computer. Would such a microcomputer implementation benefit from a particular language? Or, more precisely, to what degree does the choice of a programming language affect algorithm performance? We will examine these issues partially by comparing the computational performance of a shortest-path algorithm coded in three commonly used programming languages.

The choice of a shortest-path algorithm is due to its perserverance and simplicity. Shortest-path analysis is a major analytical component of numerous quantitative transportation and communication models. Many algorithms have been developed for finding the shortest paths from one node to all other nodes in large directed networks. Shortest-path algorithms are among the simplest mathematical programs for solving these models. These algorithms require a minimal amount of storage, can be coded in approximately 100 statements, and have been analyzed extensively.

Shortest-path algorithms have been implemented on computers, and their computational efficiency has been tested. These tests have examined the actual computer implementation of the algorithm, as well as the data-handling techniques employed in programming the algorithm. However, previous studies have focused on differences in algorithms. Our interest is in differences in language performance for a fixed algorithm.

Experimental Design

The algorithms that have been most thoroughly tested are the shortest-path algorithms, as described above. They are also the ones that require a minimal amount of storage. The specific algorithm chosen was a label-correcting algorithm examined by Pape [8].

The IBM PC running under Microsoft's disk-operating system version 2.0 was chosen for use because of the availability of software for it. It was also chosen because it is one of the standards in the microcomputer industry. The memory size on the system was 256K, although the full capacity was never used. The system had two IBM disk drives.

The languages selected were BASIC, FORTRAN, and C. Each language was included for a particular reason. BASIC is the most popular microcomputer programming language. Historically, most algorithm development has been done in FORTRAN. The majority of microcomputer implementations being done today are in C.

For each of the three languages, two versions were selected in order to investigate, for example, if the particular compiler made a difference. For C, the implementations selected were C-86 and Mark Williams. For FORTRAN, Professional FORTRAN by Ryan-McFarland (marketed by IBM) and the IBM FORTRAN Compiler version 2.0 were used. The IBM BASICA Interpreter and the IBM BASIC Compiler were the final selections.

The shortest-path algorithm was implemented in each of three languages. Each particular implementation was run through both versions of each language, resulting in two BASIC versions, two FORTRAN versions, and two C versions. In addition, to show the effect of a compiler option, the C version was compiled twice with the Mark Williams Compiler using the one register variable and the two register variable options, respectively.

Fifteen randomly generated shortest-path problems were used to test each language option. The problem sizes ranged from 100 nodes with 500 arcs to 1000 nodes with 2000 arcs. Arc lengths varied from 100 to 5000.

The central processing time was used as a measurement of each code's performance. For many applications, such as real-time routing and scheduling of vehicles, the solution time is an important factor that limits the usability of an algorithm. In the case of microcomputers, processing speeds are very slow in general. This makes solution time a very important factor when implementing mathmatical algorithms.

Table 1 contains the solution times for each of the seven executable versions on each of the 15 test problems. Interpreted BASIC exhibited considerably longer execution times than either of the compiled versions. The magnitude of the difference between solutions times using interpreted BASIC and the compiled programs can be attributed to the characteristics of interpreters. With an interpreter, if a program performs a subroutine many times, it will be translated over and over again many times. With a compiler, on the other hand, the translation is done only once. The summary statistics (in particular ratio/FTN—the ratio of the average solution time to that of FORTRAN) indicate that the particular compiler implementation may be more important than the actual language employed. That is, although C-86 performed better than the IBM FORTRAN Complier, the Professional FORTRAN times are better still. A final remark is that althought the register variable (rv) option results in impressive times with the Mark Williams complier, it prevents a "fair" comparison with C-86.

EXPECTED TRENDS

Microcomputers now provide the power previously associated only with larger, far more expensive computers. IBM's PC AT and PC XT, Digital Equipment Corporation's MicroVAX II and UNIX-based supermicros offer minicomputer performance at microcomputer price levels. With networking and further improvements in hardware technology, microcomputers will compete with small mainframes in processing performance.

Users at group or department levels will want to use this power to set up multiuser information systems. However, to implement these systems without mainframe-sized budgets, users and major independent software professionals are turning to new fourth-generation languages (4GLs) designed for microcomputers. The emergence of faster microprocessors and these more efficient languages (4GLs) could increase the use of microcomputers substantially as a tool for solving problems. With such a scenario, traditional measures of language performance become less important and are replaced by less easily quantifiable ones, for example, ease of use, functionality, and so forth.

REFERENCES

1. P. M. Hansen, M. A. Linton, R. N. Mayo, M. Murphy, and D. A. Patterson, "A Performance Evaluation of the Intel iAPX432," *Comp. Archit. News*, 10(4), 17–26 (June 1982).
2. R. D. Grappel and J. E. Hemenway, "A Tale of Four Microprocessors: Benchmarks Quantify Performance," *Electron. Des. News*, 179–365 (April 1, 1981).

3. W. Patstone, "16-Bit-Microprocessor Benchmarks–An Update with Explanations," *Electron. Des. News*, 169–171 (September 16, 1981).
4. Dennis Kneale, "Which Computer IS Speediest?" *Wall Street J.*, Friday, April 4, 1986, p. 21.
5. Donald E. Knuth, "An Empirical Study of FORTRAN Programs," *Software—Pract. Exper.*, 1, 105–133 (1971).
6. "A Guide to Langauage Performance," *PC Mag.*, 128–141 (September 1983).
7. Avram Tetewsky, "Benchmarking FORTRAN Compilers," *Byte*, 218–224 (February 1984).
8. U. Pape, "Implementation and Efficiency of Moore-Algorithms for the Shortest Route Problem," *Math. Programming*, 7, 212–222 (1974).

RICHARD S. BARR
LAWRENCE M. SEIFORD

BIBLIOGRAPHIC CONTROL OF MICROCOMPUTER SOFTWARE

INTRODUCTION

This article describes the introduction into our culture of a new form of knowledge production and recording. The particular format, computer software, for these activities is no longer new, however, and has existed for at least 30 years; but it is important to acknowledge that librarianship has had little concern with knowledge held in this format except for the recent past. This article will deal only with the period beginning in 1981 and ending in 1986.

This period of time is of interest because computer software, as tools of knowledge production and recording, became diffused in Western culture to a degree not observed during any prior time in the history of computing machinery. Moreover, there is evidence that the greatest rate of expansion and diffusion of these tools has already passed and hereafter will increase at a slower and more predictable pace.

It is much easier to document the beginning of this process of diffusion in 1981 than its state in the year 1986. No one at this time, for example, knows with any certainty the total number of discrete computer software packages available to the public for purchase or free use. Furthermore, no one knows, with any certainty, the number of individuals now employing these tools as an integral part of their daily activities. These two numerical indicators may only be estimated, and with very low degree of precision.

In 1981, through its annual publication, *U.S. Industrial Outlook*, the Department of Commerce estimated the sale of approximately 800,000 microcomputers worth $1.6 billion, where a microcomputer was conveniently described as any computer costing less than $10,000. The same publication estimated the value of software sold for these devices in 1981 at about $200 million. By 1983, the value of software sold for these devices was estimated by the Department of Commerce as having enjoyed a 10-fold increase to nearly $2 billion. In 1986, the Department of Commerce estimated that nearly $4 billion of software was sold, together with 5 million microcomputers, worth approximately $12 billion. But this rate of growth has slowed considerably since 1984, and annual growth in software, at least, is expected to remain at about 15% per year through 1990.

This flattening of the rate of growth in sales of these machines and supporting software strongly affected the many individual firms involved with this industry. The Department of Commerce also estimated that the number of personal computer vendors fell from 200 in 1983 to 150 in 1985 and that the number of major software-producing firms dwindled from 200 to 50 during this same period. (Additionally, during these same 2 years, 110 computer magazines ceased publication.)

Like any other physical entities that bear knowledge and are used by the public, such as books and journals, microcomputer software packages become a part of the network of knowledge constructed in the practice of bibliography. But the character of bibliographic practice is defined by its authors, and in the case of microcomputer software, the authors of its bibliography arose and exist outside traditional scholarly institutions. To the extent that the public requires libraries to provide access to the network of information regarding microcomputer software (if not access directly to the works themselves), libraries are affected by these instruments of bibliographic control. Because these instruments are wedded to instrumental use and economic viability, they too have been influenced by developments in the industry of computing machinery. The next section of this article deals directly with the growth in number and scope of these instruments during the period from 1981 to 1985.

BIBLIOGRAPHIC CONTROL OF MICROCOMPUTER SOFTWARE

It is probably impossible to state the number of unique and discrete software packages available in 1981, though surely it must have been considerably less than 10,000, because the total number of packages listed in the major directories of microcomputer software available in that year do not total that figure, and some overlap, however small, would be expected among these finding tools. Estimates of available software packages for public purchase or use in 1986 vary widely, but popular sources tend to converge on a figure of approximately 50,000 individual packages.

Table 1 lists 34 microcomputer software directories arranged alphabetically by year of introduction. This list, although by no means comprehensive, is drawn from numerous sources [*1–4*] and provides a coherent picture of the development of these bibliographic tools. (It should be observed, however, that one reference [5] lists 300 sources of software available during the period from 1980 to 1984; many sources of software other than microcomputer directories are noted in this work, whereas the list in Table 1 is concerned with microcomputer directories alone.)

There are many observations that can be made from this list, but one of the more striking ones is surely that there are almost no new listings of directories after 1984. Indeed, almost all new publications of directories after this period are not "new" publications in any sense and are merely industry-specific versions of larger general directories, such as the specialized publications of Datapro Research Corporation and Elsevier Science Publishing (this development will be treated more fully below). Moreover, several directories that were in print in earlier years ceased publication by 1985.

The list in Table 1 also reflects the varying hardware dominance attained by various manufacturers and producers of operating system software over the years in the period. For example, the early entries are dominated by Apple devices and CP/M operating systems, whereas later entries reflect the dominance of IBM systems and compatible machines, as well as other, lesser market survivors such as Commodore and Atari.

Figure 1 graphs the growth in microcomputer sales against the growth in the number of firms publishing microcomputer software directories. The appearance of tools published by these firms reflects the growth of the general microcomputer software industry. During the years when the rate of industry growth was greatest (from 1982 to 1984), 28 separate directories appeared. But as this growth declined, and consolidation occurred among

TABLE 1 Microcomputer Software Directories Published since 1980

Publisher	Title	First issue	Approx. No. of entries	Market	Price ($)	Notes
Addison-Wesley	Book of Apple Software	1980	500	Apple	19.95	
Advanced Software Technology	Van Love's Apple II–III Software Directory	1981	1,000	Apple	19.95	OP 1984
Cambridge Information	CP/M Software Directory	1981	1,650	CP/M	59.50	
Datapro Research	Datapro Directory of Microcomputer Software	1981	6,000	General	375.00	
Auerbach Publishers	Auerbach Computer Technology Reports: Microworld	1982	500	General	295.00	
Elsevier Science	The Software Catalog	1982	31,000	General	153.00	Online ed.
Howard W. Sams	Commodore Software Encyclopedia	1982	1,000	Commodore	9.95	
Information Sources	Sourcebook—Small Systems Software and Services	1982	3,000	General	125.00	Online ed.
Online, Inc.	Oneline Micro-Software Guide and Directory	1982	1,000	General	40.00	Online ed.
PC Clearinghouse	PC Clearinghouse Software Directory	1982	21,000	General	39.95	OP 1984
Que Corporation	IBM PC Expansion & Software Guide	1982	1,500	MS-DOS	19.95	
WIDL Video	Blue Book Series	1982		Apple	24.95	
Computing Publications	International Directory of Software	1983	5,000	General	285.00	
E. Arthur Brown	Timex–Sinclair Directory	1983	95	Sinclair	5.95	OP 1984
Microbanker, Inc.	Microbanker Software Directory	1983	500	Banking	75.00	OP 1984
Que Corporation	CP/M Software Finder	1983	2,000	CP/M	14.95	OP 1985
Radio Shack Education	TRS-80 Educational Software Sourcebook	1983	600	TRS-80	9.95	OP 1984
Redgate Publishing	LIST	1983	4,000	General	24.95	OP 1985
R. R. Bowker	The Software Encyclopedia	1983	22,000	General	75.00	
UNESCO	International Inventory of Software Packages in the Information Field	1983	188	General		
WIDL Video	Blue Book Series	1983		IBM	24.95	
Zenith Data Systems	Zenith Data Systems Software Directory	1983	250	Zenith	25.00	
Bank Administration Institute	Bank Microcomputer Hardware and Software Directory	1984	150	Banking	15.00	
Cahners Publishing	EDN–Microcomputer Operating Systems Directory	1984	80	Operating Systems	4.00	Single Issue
Crown Publishing	The Free Software and Catalog Directory	1984	5,000	CP/M	9.95	
Eastman Publishing	Software Buyers Guide	1984		General	150.00	Online ed.
Intel Corporation	Intel Yellow Pages	1984	3,000	Intel-based products	FREE	
International Computer Programs, Inc.	ICP Software Directory	1984	8,000	General	95.00	
Meckler Publishing	Software Publishers' Catalogs Annual	1984		General	97.50	

(Continued)

TABLE 1 (Continued)

Publisher	Title	First issue	Approx. No. of entries	Market	Price ($)	Notes
Nolan Information Management Services	Micro Software Report	1984	300	Library Services	60.00	
PC Software Interest Group	Directory of Public Domain Software for the IBM PC	1984	2,000	IBM	4.95	
Point	Whole Earth Software Catalog	1984	500	General	17.95	
Technique Learning Corporation	USMI Market Directory	1984	5,000	General	295.00	OP 1985
Reston Documentation Group	AT&T Computer Software Guide	1985	1,500	AT&T PC6300		

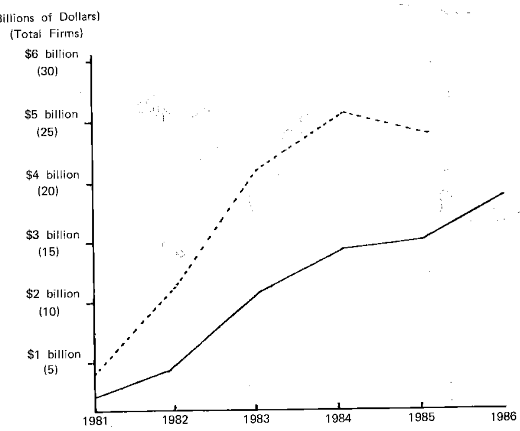


FIGURE 1 Graph showing the annual growth in microcomputer software sales (solid line) contrasted with the cumulative number of firms (dashed line), initiating publication of microcomputer software directores and maintaining those publications through the same years. By the end of 1985, eight firms had withdrawn their works, gone out of business, or substantively altered their focus away from directory publication. *Source of software sales information*: Department of Commerce, *U.S. Industrial Outlook*, annual series from 1980 to 1986.

hardware and software producers, the growth of these publications also declined, and with some consolidation as well. (R. R. Bowker, for example, a long established publisher of reference works for libraries, purchased three existing directories to add to their holding.) In a very real sense, the bibliographic apparatus of these intellectual works was shaped and controlled by the industrial environment it served.

Further evidence of this bibliographic reflection of industrial development, most notably the shrinking number of software producers, may be found in the degree of overlapping coverage among three major directories listed in Table 2. The three directories in this sample (*The Software Catalog, The Software Encyclopedia,* and *ICP Software Directory*) are all major publications, addressing the market for microcomputer software comprehensively and generally. The two largest of these directories (*The Software Catalog* and *The Software Encyclopedia*) enjoyed a 72% overlap for the sample of educational software, whereas even the smaller one (*ICP Software Directory*), which contains only one-third the number of entries as the larger ones, had 32% overlap for the sample. These results are in decisive contrast to an earlier study conducted in 1983 [*6*], which found

TABLE 2 Degree of Overlapping Coverage among Three Major Software Directories[a]

Software	Elsevier	Bowker	ICP
ACHIEVEMENT TEST—SPANISH	X		X
ALGEBRA PACK OF THREE	X	X	X
ANTONYMS & SYNONYMS—SPANISH BEGINNER	X		X
ATI FOR SPELL BINDER	X	X	
BAKE & TASTE	X	X	
BASIC CONCEPTS OF ELECTRICITY SERIES	X		
CACTUS GRADE BOOK	X	X	
EDGAR ALLEN POE	X	X	X
ELEMENTARY VOLUME 4—MATH/SCIENCE	X	X	
ESSENTIAL IDIOMS	X	X	
GENERAL CHEMISTRY II	X	X	
INTRODUCTION TO APPLESOFT BASIC	X	X	
JOB SURVIVAL SERIES	X	X	X
LANGUAGE ARTS PACKAGE	X	X	X
LEARNING AND PRACTICING WITH DISCOUNTS	X	X	
LPS: A CATALOG CARD PROG. v. 3.0	X		
MULTIPLICATION FACTS PRACTICE	X	X	
NATIONAL FLAGS	X		X
PC PILOT v. 1.41	X		
PIRATES	X	X	
REPRODUCTION ORGANS	X	X	X
SIGNED NUMBER ARITHMETIC PACKAGE	X		
SUBTRACTION PROCESS	X	X	
USAGE BONERS SERIES	X	X	
WEBSTERS NUMBERS	X	X	

[a]The percentage of overlap among the latest editions (from 1986) of three directories. The original 25 titles were sampled from the "education" applications section (pp. 2255–2433) of *The Software Catalog* (Elsevier) by selecting the last entry on every seventh page. These titles were matched against *The Software Encyclopedia* (Bowker) and the *ICP Software Directory* (International Computer Programs), yielding a 72% overlap of Elsevier with Bowker, and a 32% overlap of Elsevier with International Computer Programs.

only a 50% overlap for a small sample of vendors among three of the most popular and comprehensive publications (*LIST*, *The Software Catalog*, and *PC Clearinghouse Software Directory*), two of which (*List* and *PC Clearinghouse Software Directory*) are now out of print. Table 2 indicates a much more stable state of affairs among these works in 1986, just as the industry of microcomputing enjoyed a more stable economic status during the same period.

The next section of this article describes the development of the structure and organization of these directories, as well as the components of the main entry maintained by them. The pattern of development among these works, however, again reflects the economic influence noted above.

DEVELOPMENT AND STRUCTURE OF MICROCOMPUTER DIRECTORIES

Much of the bibliographic apparatus used throughout the Western world is the product of generations of study and thinking. That many of these tools have come to be structured in a similar manner is a result of the inherited training and practice that their editors receive during years of apprenticeship. But the systematic description and listing of computer software is a relatively new practice in bibliography in general, so that when the first waves of new works appeared in 1980 and 1981, many of the editors who established new directory publications for microcomputer software did so in innocent isolation of bibliographic training.

Prior to 1980, there were very few directories that addressed computer software, whether for large or small machines. These publications were generated by a small number of research firms that had served the current awareness product needs of data processing professionals since the middle 1960s. Some of the larger firms among these were Datapro Research, Auerbach Publishing, and International Computer Programs. But these works were organized for the service of highly trained personnel—persons who possessed years of experience with computing machinery and who were quite comfortable with the arcane and jargon-ridden vocabulary of computer systems and software.

Microcomputer software, on the other hand, has always been directed toward a mass market of users who had little in common with the specialized users of the existing software reference tools. This tension between specialized product description and general users has not yet been dissipated in these works and has posed a major problem for the original editors of works listing microcomputer software.

Another consideration in the assessment of these bibliographic instruments is that the listing and description of these materials arose after the widespread implementation of common tools supporting on-line searching. The Elsevier publication, *The Software Catalog: Microcomputers*, for example, was not initiated by Elsevier but by a small entrepreneurial firm in Colorado, who introduced the product *only* in an on-line version in 1982 through Dialog Information Services as .*Menu The International Software Database*. (Thus, Elsevier inherited a unique, completely computer-generated and serviced product and was able to impose the discipline of systematic organization on this material more easily.) In all, four firms (Elsevier, Information Sources, Online, and Eastman) would simultaneously introduce both print and on-line versions of their directories. (It should also be noted that the Source and Compuserve, two microcomputer-oriented computer

time-sharing services, also offer software listings. Additionally, there are a host of local user-supported, bulletin-board-type services providing, as of 1984, 405 separate location points for software use, purchase, review, and description [*3*].)

All in all, the initial directory publications in this field left much to be desired in terms of ease of use. The section entitled "Guide to Using this Book," from the 1981 edition of Van Love's *Apple II/III Software Directory*, provides a glimpse of some of the potential difficulties that awaited the user.

> Software is divided into main subject categories. Business, Word Processing, etc., refer to pages listed for descriptions of programs pertaining to that heading. In front of each subject section is located an index of that section. In some categories there are two sections, check both sections for listing, ex. Educational programs will have the E- with a number, E-19 which simply tells you Educational section page 19 . . . . Programs are listed by TITLE: Be sure to look carefully through each section. Some titles may be misleading or don't describe what the program actually does . . . .

In general, these early works placed a fundamental and nearly complete reliance on single-term broad subject classifications described as "applications." Even though a software package might be applicable to several types of uses denoted by the applications, it appeared under only one category. There were no syndetic structures of any kind, and users were forced to examine every entry in a section before deciding that it was not the section that described their interest. Moreover, as noted earlier, these application headings themselves represented attempts to rationalize the disparities between the technicalities of software description and the thought processes of general users. The complexity of this sort of task has been considered often [*7*] in the theoretical literature of cataloging, and the early editors of these publications should be praised for their brash confidence if for nothing else.

Unified, interfiled listings of products by titles were rare among these works, and often users were required to make multiple look-ups throughout the work to locate known titles. The first publication of *LIST* (Spring 1983 Vol. 1, No. 1), for example, provided title access by hardware system but gave no reference page for the title, so even if a user found that a product of interest was described, it would be necessary to look under all potential applications headings to find the main entry (which, when the title was sufficiently vague, could mean looking under *every* application heading.)

The main entries found in these early works vary widely in depth, quality, accuracy, and completeness. Overall, the main difficulty in these works was that the editors were forced to rely completely on descriptions and subject assignments provided by the software producers themselves. Often, software package descriptions strongly resembled sales material and seldom provided balanced feature presentation or comparative information. But in the turbulent period from 1981 to 1984, any information was often better than none in the eyes of the potential consumer. This situation has changed considerably since 1984, again mirroring the consolidation and concentration of software production in the hands of fewer, more sophisticated firms. Even in works such as *The Software Catalog* (Elsevier), where entry information is still provided by firms directly, it is apparent that the excessive self-promotion that was so evident in earlier entries has diminished.

The standard organization of bibliographic instruments follows from tradition [*7*], theoretical treatments [*8*], and empirical observation [*9*]; however, in general, the basic pattern is the location of the main elements describing a specific item in one location, with references to it from other convenient points. Use of a subject authority with controlled syndetics is preferred in all cases, but lacking that tool (and many fields lack such tools), the general rule is to be as specific as possible.

The enumeration of the elements comprising the main entry and their composition for all bibliographic instruments has engendered much debate in bibliography throughout the years. Recently, reflective of the growing interest in librarianship in computer software as collectible items, several publications have appeared that provide direction in the treatment of these materials [*10–12*]. The seminal work of S. Dodd [*12*] has been particularly useful in this effort.

Table 3 compares five directories (*The Book of IBM Software, Datapro/McGraw-Hill Guide to IBM Software, ICP Software Directory: Microcomputer Systems, The Software Catalog: Microcomputers,* and *The Software Encyclopedia*) available in 1986. All of these works represent economically stable, mature efforts to approach the problem of bibliographic control of microcomputer software. Significantly, these directories are published by or are subsidiaries of major companies that traditionally have served libraries or data processing communities, or both. Only the last three of these works are comprehensive and general ones, whereas the first two provide examples of machine-specific focus. One of these, the *Datapro/McGraw-Hill Guide to IBM Software* is an example of a market-focused publication generated from the holdings compiled for Datapro's larger, more comprehensive publication services.

The criteria in Table 3 for consideration of types of indexes are drawn mainly from discussions in Refs. 4 and 13, whereas the criteria for entry composition are drawn directly from Ref. 12.

Figure 2 displays sample entries from all five publications in Table 3. These entries present very complete treatments of these works and represent the maturity of presentations found in nearly all the remaining major directories that provide access to microcomputer software.

Figure 3 provides examples of subject headings for the field of "education," as listed in the five directories of Table 3. These examples indicate the need for some additional development of subject access in these works. The generality present in the subject headings of the first two works (*The Book of IBM Software* and *Datapro/McGraw-Hill Guide*) may be excused, to some degree, due to the relatively small number of packages available through these terms. This also holds true, though to a lesser degree, for the entries provided in the *ICP Software Directory: Microcomputers*. In this case, at least, the subject headings exhibit the application of the practice of term subordination and honestly reflect the institutional market (as opposed to the broader consumer market) focus of this work as a whole. But the headings for *The Software Catalog* seem altogether too broad for the nearly 7,500 programs to which they refer (nearly 600 per entry) and would pose considerable problems for users without the numerous indexes provided in the same work. Only the subject heading structures for *The Software Encyclopedia* seen adequate for access to this area of knowledge.

TABLE 3 Comparative Chart of Five Microcomputer Software Directories

Title	*The Book of IBM Software 1985*
First year published	1984
Price	$19.95
Orientation	All IBM machines and MS-DOS clones
Number of packages	Approx. 500
Types of indexes	
Applications	Yes
Product name	Yes
Vendors	Yes
Computer system type	No
Computer language type	No
Operating system type	No
Microprocessor type	No
Key word index	No
Subject index	Yes
Other	None
Entry composition	
Source of entry	Paid reviewer
Language required	Yes
Memory required	Yes
Additional software required	Yes
Additional hardware required	Yes
Documentation notes	Yes
Number of subject headings assigned to entry	1
Number of access points	4
Organization of the work	Main entry, including review, accessed through table of contents and indexes by page number.
Clarity of instructions	Excellent
Notes	As part of front matter, contains detailed software integration and compatability charts.
Publisher	Arrays, Inc., 11223 S. Hindry Ave., Los Angeles, CA 90045

TABLE 3 (Continued)

Title	*Datapro/McGraw-Hill Guide to IBM Software*
First year published	1984
Price	$22.95
Orientation	All IBM machines and compatibles
Number of packages	Approx. 1000
Types of indexes	
Applications	Yes
Product name	Yes
Vendors	No
Computer system type	Yes
Computer language type	No
Operating system type	No
Microprocessor type	No
Key word index	No
Subject index	No
Other	Includes detailed vendor report, including financial status and maturity
Entry composition	
Source of entry	Datapro editors
Language required	Yes
Memory required	Yes
Additional software required	Yes
Additional hardware required	Yes
Documentation notes	Yes
Number of subject headings assigned to entry	1
Number of access points	3
Organization of the work	Main entry access provided through table of contents with brief title entry in other indexes.
Clarity of instructions	Excellent
Notes	Drawn from the general *DataPro Directory of Microcomputer Software*, a comprehensive software information service with many custom features, available for about $400 per year. Other publications in this series include those for CP/M and Apple systems.
Publisher	Datapro Research Corporation, 1805 Underwood Blvd., Delran, NJ 08075

(Continued)

TABLE 3 (Continued)

Title	*ICP Software Directory: Microcomputer Systems*
First year published	1984
Price	$95
Orientation	General
Number of packages	Approx. 8,000
Types of indexes	
Applications	Yes
Product name	Yes
Vendors	Yes
Computer system type	Yes
Computer language type	No
Operating system type	No
Microprocessor type	No
Key word index	Yes
Subject index	No
Other	None
Entry composition	
Source of entry	Vendor (edited by ICP)
Language required	Yes
Memory required	Yes
Additional software required	Yes
Additional hardware required	Yes
Documentation notes	Yes
Number of subject headings assigned to entry	1
Number of access points	5
Organization of the work	Main entry by catagory (e.g., application), with brief entry at other access points.
Clarity of instructions	Excellent
Notes	ICP, the world's oldest publisher of software information, began business operations in 1966. The company also offers directories to mainframe and minicomputer software. This work is published in two volumes.
Publisher	International Computer Programs, Inc., 9100 Keystone Corssing, P.O. Box 40946, Indianapolis, IN 46240

TABLE 3 (Continued)

Title	*The Software Catalog: Microcomputers, Summer 1986*
First year published	1983
Price	$75 per issue
Orientation	General
Number of packages	Approx. 31,000
Types of indexes	
Applications	Yes
Product name	Yes
Vendors	Yes
Computer system type	Yes
Computer language type	Yes
Operating system type	Yes
Microprocessor type	Yes
Key word index	Yes
Subject index	Yes
Other	None
Entry composition	
Source of entry	Vendor
Language required	Partial
Memory required	Yes
Additional software required	Partial
Additional hardware required	Yes
Documentation notes	No
Number of subject headings assigned to entry	2
Number of access points	9
Organization of the work	Main entry available alphabetically by vendor number prefix with brief entries elsewhere. Key word index lists package number only.
Clarity of instructions	Excellent
Notes	Various issues contain glossaries and other explanatory and review material. Other services provided include specialized bibliographies produced on demand and a software-ordering service. The entire file is also available through DIALOG.
Publisher	Elsevier Science Publishing Co., Inc., 52 Vanderbilt Ave., New York, NY 10017

(Continued)

TABLE 3 (Continued)

Title	*The Software Encyclopedia 1985/1986*
First year published	1983
Price	$75.00
Orientation	General
Number of packages	Approx. 22,000
Types of indexes	
Applications	Yes
Product name	Yes
Vendors	Yes
Computer system type	No
Computer language type	No
Operating system type	No
Microprocessor type	No
Key word index	No
Subject index	No
Other	None
Entry composition	
Source of entry	Vendor
Language required	Yes
Memory required	Yes
Additional software required	No
Additional hardware required	No
Documentation notes	No
Number of subject headings assigned to entry	1
Number of access points	3
Organization of the work	Main entry available through both application and product name indexes. Brief entries are provided through vendor index.
Clarity of instructions	Excellent
Notes	Subsumes earlier publications of Dekotek (*Microcompuer Marketplace*), PC Telemart (*PC Clearinghouse Software Directory*), and Advanced Software Technology (*Van Love's Apple II–III Software Directory*).
Publisher	R. R. Bowker Company, 205 E. 42nd St., New York, NY 10017

WORDSTAR MENU POWER and WORDSTAR COMMAND POWER

American Training International
$75.00 each
128K

Overall Rating	B	B	*Reliability*
Ease of Use	A	B	*Error Handling*
Documentation	B	C	*Value for Money*

These programs familiarize you with the menu structure and command features of *WordStar*. They come with manuals and working tutorial disks. The manuals paraphrase the regular *WordStar* manual excellently but are much simpler to use and read. They include a useful chart called the "Menu Power Roadmap" that shows you a diagram of the seven *WordStar* menus and a chart listing of the function commands. The training disks take you step-by-step through each of the menus and commands.

The ATI manual states that *Menu Power* will take you less than an hour to complete, after which time you should feel reasonably familiar with the *WordStar* menus. *Command Power* takes from three to four hours to complete. After you have finished both programs, you should feel comfortable enough with *WordStar* to get it up and running quite successfully. You will probably need to refer to the manuals only for more complicated functions. After using the ATI manuals with their excellent format, simplicity, and ease of use, you will probably avoid the MicroPro manual except for extremely sophisticated functions. The ATI manual deliberately excludes them and refers you to the MicroPro manual in such instances.

Having used *WordStar* already, I figured that I wouldn't learn much from the ATI training session. Much to my surprise, I found myself not only learning at each step, but enjoying the exercises laid out by the tutorial. I also found that the simplified ATI manuals made previously incomprehensible things clear. What a discovery! Most of the people who write documentation for software need to take lessons from the staff at ATI. The ATI manual explains virtually every section more simply than the original.

These training programs could be worth a great deal to the proper audience. Small businesses, especially, should keep them in mind.

Compatibility:
Unknown

1) *The Book of IBM Software*

Citation

Eagle Enterprises
2375 Bush Street
San Francisco, CA 94115
(415) 346-1249

Operating environment: 56KB—96KB RAM.
Source language name and user availability: Basic.
Usage pricing: a purchase price of $185.
Maintenance provisions/pricing: a 30-day warranty.
Documentation: user's manual included.
Physical medium: 5¼" or 8" diskette, hard disk.
Product description: Citation provides for the storage of factual information with simultaneous cross-referencing by subjects of interest, and selective retrieval based on interrelationships of the stored items. This system has its own on-screen text editing and provides a keyword index report. Other reports include a file listing, keyword list and multi-keyword selection report.

2) *Datapro/McGraw-Hill Guide to IBM Software*

FIGURE 2 Entry samples for the five directories listed in Table 2.

LI'L CHIEF GENERAL LEDGER
Product Type: Applications Software
Geographic Area Served: Worldwide
Hardware Supported: IBM PC, PC-XT, PC-AT; AT&T 6300/7300; Columbia; COMPAQ; Wang PC; Hewlett-Packard Series 100; Zenith; Radio Shack; Altos; TeleVideo; KAYPRO; MS-DOS-based Hardware; UNIX-based Hardware
Operating Systems: PC DOS, MS-DOS, CP/M-80, TurboDOS
Languages: Not Specified
Number of Clients/Users: 1,000
Narrative: The Li'l Chief General Ledger module is a standalone, general-purpose system designed for home users and small businesses. It features a user-definable chart of accounts using 12-character, alpha-numeric accounts and account scheduling. It provides for current and comparative balance sheets; income statements (both departmental or combined) and ratio reports. Also included are edit lists, journal summaries, registers, trial balances, activity and account detail reports.
Special Configuration Requirements: 64K RAM

Contact Data	**Pricing**
Thoughtware Publishing Company	$175.00
P.O. Box 669	
Grants Pass, OR 97526	
Tele. 503-476-1468	*P27916*

3) *ICP Software Directory: Microcomputer Systems*

WORD **ISPN 53150-750**

Features word wrap, footnotes, subscripts, superscripts, undo command, style sheets (preformatted set of options) and more. Features include: direct formatting, on-line help system, multiple windows, glossary buffers, multiple fonts and formats and a horizontal scroll for text that is wider than the screen. The direct formatting feature allows the user to designate characters, words, paragraphs or series of paragraphs to be boldfaced, italicized, underlined or struck through. The form letter, or merge, facility allows the user to produce custom form letters by merging variable information with a standard letter. The system merges a primary document with data stored in another document or in an ASCII file.
Released 1/85, *reviews available*; *subjects*: 187. *Systems*: IBM PC, IBM PC AT, IVY MICROCOMPUTER 3001, IVY MICROCOMPUTER 3002, RADIO SHACK TANDY 2000, CP/M (DIGITAL RESEARCH), MSDOS (MICROSOFT), 5-1/4-inch disk, $375.

4) *The Software Catalog: Microcomputers, Summer 1986*

Learning & Practicing with Money.
Compatible Hardware: Apple II, APF Imagination Machine, Xerox 820, Osborne, TRS-80, Commodore, IBM, Kaypro.
Operating System(s) Required: CP/M.
Price Info.: APF Imagination Machie, Xerox 820, Osborne, TRS-80, Kaypro disk $34.95; Apple II, Xerox 820, IBM, TRS-80 8" disk $34.95; APF Imagination Machine cassette tape $34.95. Resource Software International, Inc.

5) *The Software Encyclopedia 1985/1986*

FIGURE 2 (Continued)

Education 435
Computer Literacy 436
For Young Children 449
Computer Aided Instruction 452
Mathematics 456
Reading, Spelling, and Language Skills 461
Typing 466
Miscellaneous 468

1) *The Book of IBM Software 1985*

Education
Administrative Business Information Systems 116
Authoring Systems 116
Computer Assisted Instruction 117
Computer Training 121
Student Administration & Scheduling 128

2) *Datapro/McGraw Hill Guide(s)*

63. ...... **EDUCATION** **394A**
63.1 .... **Educational Institutions**
63.1.1 .... **Integrated Administrative Management** **395**
63.1.2 .... **Accounting Support** **397**
63.1.3 .... **Funding/Loans/Scholarships** **401**
63.1.5 .... **Scheduling** **401**
63.1.6 .... **Records** **403**
63.1.7 .... **Student Information Management** **405**
63.1.8 .... **Classroom and Grading Administration** **407**
63.1.9 .... **Other Administrative Services** **417**
63.2 .... **Instructional Support Systems** **419**
63.3 .... **Learning Aids** **432**
63.4 .... **Libraries** **592**

3) *ICP Software Directory: Microcomputers*

200 EDUCATIONAL
214 Administration
231 CAI – Humanities
233 CAI – Language Arts
232 CAI – Math
234 CAI – Science
237 CAI – Social Science
235 CAI – Special Education
236 CAI – Other Basic Skills
244 CMI – Computer-Managed Instruction
250 Computer Literacy
254 Counseling/Aptitude Testing
274 Library Management
279 Miscellaneous Educational

4) *The Software Catalog: Microcomputers, Summer, 1986*

FIGURE 3 Subject classifications for "education" applications.

Educational Applications

EDUCATION

Education – Adult Education
Education – Art
Education – Art History
Education – Business – Business Management
Education – Business – Business Mathematics
Education – Business – Economics
Education – Business – General
Education – Business – Typing and Shorthand
Education – Drivers Education
Education – Financial Aid
Education – Foreign Languages – French
Education – Foreign Languages – General
Education – Foreign Languages – German
Education – Foreign Languages – Hebrew
Education – Foreign Languages – Italian
Education – Foreign Languages – Latin
Education – Foreign Languages – Russian
Education – Foreign Languages – Spanish
Education – General
Education – Grammar and Language Arts – Alphabet Recognition
Education – Grammar and Language Arts – Elementary School
Education – Grammar and Language Arts – English as a Second Language
Education – Grammar and Language Arts – Literature
Education – Grammar and Language Arts – Phonetics
Education – Grammar and Language Arts – Punctuation
Education – Grammar and Language Arts – Reading
Education – Grammar and Language Arts – Secondary School
Education – Grammar and Language Arts – Spelling
Education – Grammar and Language Arts – Vocabulary
Education – Grammar and Language Arts – Writing
Education – Guidance
Education – Health and Hygiene
Education – Home Economics
Education – Mathematics – Algebra
Education – Mathematics – Arithmetic
Education – Mathematics – Basic Skills
Education – Mathematics – Calculus
Education – Mathematics – General
Education – Mathematics – Geometry
Education – Mathematics – Statistics
Education – Mathematics – Trigonometry
Education – Music
Education – Physical Education
Education – Preschool Education
Education – Religious Education
Education – SAT (Scholastic Aptitude Test) Tutorials
Education – Science – Anatomy and Physiology
Education – Science – Astronomy
Education – Science – Basic Skills
Education – Science – Biology
Education – Science – Chemistry
Education – Science – Computer Literacy
Education – Science – Earth Science
Education – Science – General
Education – Science – Natural History
Education – Science – Physics
Education – Social Sciences – Agriculture
Education – Social Sciences – Anthropology
Education – Social Sciences – Civics
Education – Social Sciences – Culture Studies
Education – Social Sciences – Current Events
Education – Social Sciences – Economics
Education – Social Sciences – General
Education – Social Sciences – Geography
Education – Social Sciences – History
Education – Social Sciences – Psychology
Education – Social Sciences – Sociology
Education – Social Sciences – World History
Education – Special Education
Education – Typing
Education – Vocational Training

EDUCATIONAL ADMINISTRATION

Educational Administration – Authoring Systems
Educational Administration – Classroom – General
Educational Administration – Classroom – Gradebooks
Educational Administration – Classroom – Teacher Aids
Educational Administration – School – Financial Accounting
Educational Administration – School – General
Educational Administration – School Scheduling
Educational Administration – Skills

5) *The Software Encyclopedia 1985/1986*

FIGURE 3 (Continued)

Finally, an example of the growth in the simple number of packages offered in all areas of knowledge in these works is provided in Table 4. The 25 titles used in the overlap sample of Table 2 from *The Software Catalog* were traced back through four successive editions of this work. Only one title from this list was evident in the first edition of *The Software Catalog*, which first appeared in 1983.

The next and final section of this article addresses the complex question of the intellectual status of these works and the degree of attentiveness with which the profession of librarianship must ultimately regard them. The bibliographic instruments that convey access to these materials surely reflect the initially explosive and now more stable growth of this industry, and this article has done much to document these events. But the meaning of the diffusion of these works in our culture must also be examined, for if these works are regarded as truly knowledge bearing, in the same manner as books and journals are presently regarded, then librarianship must come to terms with its ancient obligation to collect these materials systematically and make them available to patrons. The discussion that follows is intended to be a brief consideration of these issues and is offered only to stimulate further discussion of them.

INTELLECTUAL STATUS OF COMPUTER SOFTWARE

The nature of the debate regarding the intellectual status of computer software is well encompassed by developments on a far more homely topic, that is, the question of whether to assign International Standard Book Numbers (ISBNs) to these works.

It is quite desirable to have an internationally recognized system for numbering and identifying works of computer software. If available, such a system would be extremely helpful in recording software sales, ordering software, and distinguishing among various versions, copies, and editions. One scheme for accomplishing this objective was initiated with the first publication of Elsevier's *The Software Catalog*. In that edition, and all succeeding ones, Elsevier assigned a unique number, classified by vendor and termed the International Standard Program Number (a trademark of Elsevier), or ISPN. But the exclusive use of this numbering system by Elsevier and its retention of trademark rights prohibits this identification system from wide use and acceptance. (Another design of this type, the Universal Software Market Identifier, used by Technique Learning Corporation to enhance the acceptance of its directory, *USMI Market Directory*, was less successful; this work is now out of print.)

The ISBN, by 1984, was already in use as a means of identifying software, but in that same year, a committee sponsored by the National Information Standards Organization (NISO) (Z39) proposed the use of a new system for identification of software, entitled Standard for Computer Software Numbering, or SCSN [*14*]. Significantly, in the context of this discussion, the objections to the use of the ISBN were not made on any intellectual grounds. The ISBN was rejected only because of the complexity of software production, that is, not only are different versions of the same product created for different machines but also for different operating systems on the same machines.

During 1985, a number of objections to the creation of a new numbering system, the SCSN, for computer software were raised [*15*]. Again, how-

TABLE 4 Growth in Number of Software Packages[a]

	1986	1985	1984	1983
ACHIEVEMENT TEST—SPANISH	X	X		
ALGEBRA PACK OF THREE	X	X		
ANTONYMS & SYNONYMS—SPANISH BEGINNER	X	X	X	
ATI FOR SPELL BINDER	X	X	X	
BAKE & TASTE	X			
BASIC CONCEPTS OF ELECTRICITY SERIES	X	X		
CACTUS GRADE BOOK	X			
EDGAR ALLEN POE	X			
ELEMENTARY VOLUME 4—MATH/SCIENCE	X	X	X	
ESSENTIAL IDIOMS	X			
GENERAL CHEMISTRY II	X	X		
INTRODUCTION TO APPLESOFT BASIC	X	X		
JOB SURVIVAL SERIES	X	X		
LANGUAGE ARTS PACKAGE	X			
LEARNING AND PRACTICING WITH DISCOUNTS	X	X		
LPS: A CATALOG CARD PROG. v. 3.0	X	X	X	
MULTIPLICATION FACTS PRACTICE	X			
NATIONAL FLAGS	X			
PC PILOT v. 1.41	X	X		
PIRATES	X	X		
REPRODUCTION ORGANS	X	X		
SIGNED NUMBER ARITHMETIC PACKAGE	X	X	X	X
SUBTRACTION PROCESS	X			
USAGE BONERS SERIES	X			
WEBSTERS NUMBERS	X			

[a]The increase in entries for the sampled titles in education, drawn from the Summer 1986 edition of The Software Catalog (Elsevier). Back issues of this directory for Winter 1985, Spring 1985, and Spring 1983 were searched for the 25 entries obtained in the sample. Remarkably, only one entry from 1986 was present in 1983, indicating significant growth in this software production catagory.

ever, no intellectual arguments were offered, and the focus of the objections was simply that another numbering system would be too difficult to implement among so independent a group as software publishers.

The conclusion to be drawn from these debates is a pragmatic one: In the opinion of observers throughout many branches of librarianship, computer software is already indistinguishable from texts and is accepted as textual in nature. What are some of the questions surrounding this conclusion?

The significance of this discussion is wholly practical because the manner in which an object is defined has traditionally determined its treatment in librarianship. Failure to define computer software as textual in nature classes these objects as cultural ephemera and distracts us from their aesthetic and instrumental uses. Librarianship does not, for example, care much about the bibliographic control of shapes of bottles, designs of automobile grills, or brands of cigarettes, and it is only the rare special library or collection that is much concerned with these matters. But it will be helpful in the tentative remarks that follow to consider texts from a teleological (the end use or purpose of things) perspective. Let us therefore consider the end uses of texts.

A valid manner in which to characterize texts is to assert that they are extensions of the minds of individuals and groups [*16*]. The mind has various capacities; among them, reasoning, imagination, and memory. When a text is consulted, in this characterization, the mind of another is being consulted. We wish to share something of the reasoning, memory, or imagination of that mind, and just as such intercourse with a living human being alters our awareness of the world, contact with texts alter our awareness. But the question of degree enters into this discussion and must also be addressed.

If someone meets a stranger on the street and asks them the time of day, is this not an example of an ephemeral contact between minds of the most inconsequential and transitory value? But the answer to this question rests not upon the character or stature of the person of whom the time is required, nor even in the accuracy of his or her answer. Rather, the answer to the question of the value of this contact lies in the mind of the enquirer. If the answer is important to the enquirer, then the question is not unimportant, inconsequential or ephemeral.

In the same manner, in librarianship, when a text is collected, assessments are made of the probability that future consultations of the text will be more for serious purposes than for trivial ones. And, whenever a text is collected, the decision has been made that the probability of future serious consultations of that text is greater than the probability of trivial or ephemeral ones. It is thus proposed that such assessments should also be made of computer software as they are presently made for texts.

It is easy to characterize computer software as an extension of the mind commensurate with texts. Software may, for example, record human reasoning directly in the form of an expert system or instructional package. It may also record human imagination, as in the case of a program that interactively constructs games. Even human memory may be encompassed to some degree by computer software, such as, for example, data base management systems.

But those who would argue against treating computer software as texts would claim that software is only a transitory extension of the human mind and is therefore not likely to be consulted for future serious purpose.

There may well be merit in this claim. For example, a computer software package that does nothing more than create banners (signs of one large letter per page) might be viewed as little more than a tool, and if a library were to collect such an item, it would then also be obligated to collect wrenches, saws, hammers, and other such tools.

But those who seize on this example to support a case against the consideration of computer software as texts would extend it to many other types of software other than packages that create banners, such as data base management systems and spreadsheet packages, which may also be considered as mere tools. But if this extension was correct, libraries would not collect programmed texts, or indeed any other textbooks, because these also are only tools, nor would they collect directories (including all those discussed in this article) and other reference works. The point suggested by this discussion is simply that the worth of an item is not determined by its format. It is the future use of the item that is important and must be assessed.

So let us conclude these remarks not by proposing that all computer software programs should be collected, but rather by proposing something far less sweeping: that computer software should be subjected only to the same evaluations as conventional texts. Moreover, at least insofar as such evaluations rest on the assessment of the probability of future serious consultation by some user or users, some computer software must also be collected. But one final point must be made. The meaning of the phrase "conventional text" must be considered, because it is no longer clear exactly what is meant by this expression. On my desk at this moment is a particular text. It is a conventional text for children in all particulars but one: It is also a computer program. The text illustrates the song entitled, "Yankee Doodle Dandy," popular in American colonial days. When the text is opened, a piezoelectric cell powers a computer program resident on a microchip to play that song. Is this conventional text a computer program? Is this computer program a conventional text?

ACKNOWLEDGMENT

The author gratefully acknowledges the assistance of International Business Machines Corporation in completing this article for its permission to use the collections of the IBM Austin Research Facility Library.

REFERENCES

1. *Gale Directory of Directories*, 1985, third ed., Gale Research Company, Detroit, MI, 1984.
2. Robert T. Fertig, *The Software Revolution*, North-Holland Press, New York, 1985.
3. Alfred Glossbrenner, *How to Buy Software*, St. Martin's Press, New York, 1984.
4. Mark E. Rorvig, "The 'Bibliographic' Control of Microcomputer Software," *Electron. Libr.*, *2*(3):183–195 (1984).
5. Ted Kruse, *Locating Computer Software*, Garland Publishing, New York, 1985.

6. "Micro Software Markets," *Comput. Age Software Dig.*, *15*(8):9 (1983).
7. Francis Miksa, *The Subject in the Directory Catalog from Cutter to the Present*, American Library Association, Chicago, IL, 1983.
8. J. M. Brittain, *Information and Its Users*, Wiley-Interscience, New York, 1970.
9. Ben-ami Lipetz, *User Requirements in Identifying Desired Works in a Large Library*, U.S. Office of Education, Washington, D.C., 1970.
10. *Guidelines on Subject Access to Microcomputer Software*, American Library Association, Resources and Technical Services Division, Chicago, IL, 1986.
11. *Guidelines for Using AACR2 Chapter 9 for Cataloging Microcomputer Software*, American Library Association, Chicago, IL, 1984.
12. Sue A. Dodd, *Cataloging Machine Readable Data Files*, American Library Association, Chicago, IL, 1982.
13. Harold Borko and Charles L. Bernier, *Indexing Concepts and Methods*, Academic Press, New York, 1978.
14. National Information Standards Organization [Z39], Memorandum of November 15, 1984, National Bureau of Standards, Gaithersburg, MD, 1984.
15. Patrick Wilson, *Two Kinds of Power*, University of California Press, Berkeley, CA, 1968.
16. E. Glynn Harmon, *Human Memory and Knowledge*, Greenwood Press, Westport, CT, 1973.

MARK E. RORVIG

BILINGUAL PROCESSORS

DEFINITION OF THE BILINGUAL PROCESSOR CONCEPT

Bilingual processors are capable of mixing the written symbols from two or more natural languages in the same document; this includes transitions from one symbol set to another in the same paragraph, line, or word. The term bilingual, then, does not apply to processors that require a chip modification to make them speak another language. Processors requiring a chip modification to switch languages speak only one language at a time and are therefore monolingual. Bilingual processors instead use alternate "keyboards," which are continually available and may be invoked whenever an alternate symbol set is needed. Because the basic procedures for handling two languages are similar to those for managing several languages, the term bilingual will be used here to refer to processors that are bilingual or multilingual.

THE VALUE OF BILINGUAL PROCESSORS

From the outset, we should answer a pair of questions that are likely to occur to the reader who contemplates the great efficiencies afforded by alphabetic writing and comes to grasp the powerful ubiquity of the "roman" alphabets that have been spread throughout the world by the influence of the Western cultures: Given these virtues of the Western alphabets, why should we be concerned with representing non-Western languages in a computer, and why not just transliterate them all to roman script?

The first part of an answer to these questions concerns our human context. Contrary to the standard cliché, everyone does not know English, and if we expand that cliché to include other Western languages, all of which have influenced many parts of the world (e.g., Dutch, Flemish, French, German, Greek, Italian, Norwegian, Portuguese, Spanish, Swedish, etc.), everyone does not know, or even read, one of those languages either. A very large proportion of the world's population speaks non-Western languages, many of which carry with them the associations gathered during thousands of years of their own civilizations. To those other people, the opacities and intricacies of their languages are not impediments to communication, as we may see them, but a treasured basis for their identities from which they will not be parted. Thus, the greater portion of the world is, to us, embedded in an intricate linguistic fabric of foreign sounds, texts, and meanings with which we must be able to deal if we are to manage ourselves successfully in relation to that greater world. As we will see, the bilingual processor provides an instrument for overcoming some of the most intransigent aspects of formulating and combining texts from differing languages in an electronic medium, whether on a small scale, or in a large, rapidly changing, internationally networked environment.

A second part of an answer to the above questions is linguistic and cultural in nature. Despite the strengths of notational efficiency and empirical explicitness enjoyed by alphabetic writing systems plus the international ubiquity of roman script, other kinds of writing systems have their own peculiar strengths that cannot be gotten by alphabetic writing or roman script. The most outstanding example is Mandarian Chinese, whose ideographic symbol system is used as a basis for writing by the various Chinese language groups and by other Oriental societies (e.g., Japanese). A person who knows Mandarin script can thus communicate in writing with speakers of other Oriental languages, even though that person may not be able to communicate orally. This universalizing characteristic of Mandarin strongly enhances its entrenched prestige in Oriental societies. As we shall see below, the adaptation of bilingual processors to Oriental scripts maintains the universalizing capacity of the Mandarin component while creating for it a wholly new notational efficiency.

Finally, in an argument that must seem circular, the capabilities of the computer bring to the bilingual processor a capacity in the management of language that is impossible by other means and, in this way, strengthen the value of the languages themselves. The computer's slavish dedication to precise remembrance and intense calculation permits, in a rapid and transparent fashion, the recall of large databases and the construction of detailed decision trees (as in deciding which possible meanings fit the context of a Japanese homophone like *tou*, which has 64 different interpretations); its capability for manipulating and permutating objects, depending on their context, allows the printed representation of complicated scripts like those of Marathi, Nepali, and other Indian languages, based on sequential phonetic input, even though in the final result, the symbols for the various sounds may be turned around and superscripted or subscripted in complex ways; its capability for representing the same phonetic text in different symbolic outputs via a simple change of screen or printer driver enables, for example, the roman and Devanagari variants of a Hindi text to be printed from the same internal text code.

From these realizations—first of the unavoidability of having to deal with the world's non-Western languages, also of their own inherent strengths, and finally, of the facilitating and enhancing instrumentality available in bilingual processors—a programmatic goal emerges in the further development of bilingual processors that is found in the following passage from Joseph D. Becker, who correctly identifies the large-scale international transmission of texts as a significant new contribution of the computer, with bilingual processors as the enabling element for the automated massive communication of international text via what is now called electronic mail:

> The telephone does not require its users to speak only English, nor does a postal system require its users to write only English. Electronic mail will not succeed as a global medium unless the text it carries is fully multilingual. To my mind that is the ultimate application for multilingual word processing.

ORIGINS

Computers began to be used to manage fragments of English at MIT toward the end of the 1950's. In association with programs designed to write and modify computer code, similar efforts evolved into word processors. In the 1970's and 1980's, coinciding with the exponential miniaturization and increas in capacity that have been constant features of computing during its "fourth

generation" of development, full-fledged word processors and text formatters were created and passed through various versions, enabling the application of microcomputers to what has become one of their main focuses of activity.

The first text processors inherited an English language bias that limited the capability of computers for handling languages other than English and for managing more than one language at a time. As computer use and manufacturing grew in non-English-speaking societies, a common tactic for adapting the computer to another language consisted of reassigning internal character codes to the characters of the new language and designing screen and print drivers consistent with those modifications. This character-oriented tactic worked well enough for the languages of the world that share with English the principle of alphabetic script (e.g., the Greek alphabet, or the Russian Cyrillic alphabet, not to mention the more English-like alphabets of Dutch, French, German, Italian, Portuguese, Spanish, etc.) and for dealing with one character set at a time; but it does not adapt as easily to the requirements of syllabic script, cannot handle the ideographic symbol systems of languages like Chinese and Japanese, and is adequate for dealing with two or more languages only when the other languages share many symbols with English.

At the same time that it stabilized the standard character codes, allowing languages with English-like alphabets to be represented easily on a computer, the ASCII convention also surrendered the extended ASCII domain to various conflicting symbolizations associated with differing proprietary designs. Bilingual processors were created from efforts taken to adapt computers to non-English-like natural languages and to overcome the combined limitations of "standard" ASCII English-like constraints and "extended" ASCII arbitrary usages. But before we look at bilingual processors themselves, let us consider the opportunities afforded and the problems posed by the different kinds of writing systems.

WRITING SYSTEMS

The writing system associated with English possesses a basically phonetic design, that is, the written symbols stand in place of the sounds of the language and are strung in a sequence that corresponds to the temporal sequence of the language's spoken sounds. Even though the "fit" between English spelling and pronunciation has become quite loose as the result of the conservation of archaic forms in the spellings of many English words, the English writing system nevertheless exemplifies the objectives and design of a phonetic writing system. Moreover, the English writing system is alphabetic (as opposed to syllabic writing), which means that each written symbol is meant to stand for one sound (notwithstanding the many obvious cases in which English spelling is not perfectly phonetic). In this, English shares with other Western languages an alphabetic design and a symbol system possessing many common elements inherited through Latin and Greek from Phoenician script. Given the mediating Latin (or Roman language) forebear of Western writing traditions, this symbol system is referred to as roman script when used to transliterate non-Western languages.

A phonetic writing system that is alphabetic possesses virtues that become even more apparent when utilizing a mechanical means for rendering language. One virtue is that all possible expressions in the language can be represented via sequential permutations of a fairly small inventory of symbols (e.g., the 26 or so symbols of Western alphabets). A practical benefit of this virtue is that the symbol set (or alphabet) can be learned, remembered, and applied quite easily. Another benefit is that it can be depicted on a keyboard while using a set of keys small enough to fit within a limited space and allow each key to be found and activated with minimum effort (in fact, via automatic habit patterns). An electronic analog to these benefits is that the elements of a small symbol set can be distinguished from each other by relatively few sequential bits within a computer byte (more will be said about this later). A second virtue of an alphabetic system is that the phonetic material of the language is represented explicitly in the same sequence in which it is spoken (Subject to obvious qualifications based on degree of phonetic fit), which then permits the system to be used to capture and portray language that has not been experienced previously, including limited foreign language expressions.

The roman alphabet adds to the above-outlined virtues of an alphabetic system the historical fact that it is well known throughout the world in its several Western-language variants, as the result of the worldwide influence of the Western cultures. Because of this ubiquity of the roman alphabet, it is always available for transliterating another language when such is desirable, and its universal familiarity frequently makes it the symbol system preferred for phonetic representation even by speakers of non-Western languages.

Syllabic writing is phonetic too, but it differs from alphabetic writing in that in syllabic writing one symbol stands for the combined sounds of a syllable, each one of which may or may not be represented by some characteristic of the symbol. Two examples of syllabic writing are Cherokee and the *hangul* system devised for writing the native language of Korea (different from the script used in official and scholarly Korean writing, which is Mandarin). Some of the Cherokee symbols and their phonetic referents are H = *mi*, Z = *no*, K = *dzo*, G = *nah*, and so forth (I have chosen only English-like symbols, because they are easier to represent here—many unusual symbols also exist in the Cherokee syllabary, which was influenced, nevertheless, by English). There are very few true syllabic writing systems among the world's languages, and many have been abandoned in favor of alphabetic systems; an important exception is Japanese, which combines Mandarin-derived ideograms (called *kanji*) with a syllabary (called *kana*, composed of about 50 Japanese characters) that is used to represent the complicated grammatical endings of Japanese words.

Ideographic writing systems, like Chinese, must use thousands of distinct symbols to represent the many different word meanings found in the language. Such a system requires a large investment of time in learning how to read and write the language and creates monumental difficulties for its mechanical typing (until the recent inception of the word processor coupled with bilingual processing techniques, a professional typist for Japanese had to be a physically strong person who could produce about 20 characters per minute, which amounts to approximately 10 pages per day).

Writing systems also differ in the order in which symbols are placed on a page. The convention common to Western languages is that the alphabetic symbols are strung from left to right in rows that are read from the top to the bottom of the page. Arabic and Hebrew, on the other hand, are written from right to left but also from top to bottom. The ideographic symbols of Chinese and Japanese are traditionally strung in columns, from top to bottom, and the columns are read from right to left; however, modern Asian printing

practices string the characters in rows from left to right. Mongolian, which is traditionally strung in columns from top to bottom and read from left to right, is rotated by 90 degrees for combination with horizontal text.

From this brief discussion of the various kinds of writing systems, it can be appreciated that the complexities of representing two or more languages via the same text processor go beyond those that can be solved by adding a few special characters to a common alphabet or even by replacing one symbol set with another. A text processor that can truly manage any of the world's languages must be able to combine the distinct symbolic strategies of ideographic systems and phonetic systems, must be able to mix texts from languages that order their written symbols in different ways, and must be able to call on many different screen and print drivers in order to output the radically differing symbols of disparate languages. In this, it should be mentioned that the physically impossible is not attempted—there is no way that horizontal and vertical text can be commingled freely, although horizontal text written from left to right can be mixed with horizontal text written from right to left, and the conventions of modern Oriental print allow the conversion of vertical text into a horizontal equivalent.

What is not readily apparent in the above discussion is that the screen and print driver solutions that allow the visual representation of the differing symbols of many languages are the same kinds of solutions that permit the representation of the varying visual characteristics of different fonts and font families. For that reason, there is a very natural marriage of bilingual processors with what has recently come to be called "electronic publishing", and also "desktop publishing", a consideration that will be illuminated further in (Input and Output Solutions), below.

SYMBOL DISTINCTION ENCODING SOLUTIONS

The ASCII standard is based on an 8-bit character (or byte, also confusingly called a computer word), seven of whose bits are available for symbol coding, thus permitting 2^7, or 128 symbols; in extended ASCII, when the eighth bit is available for symbol coding, 2^8, or 256, symbols are possible. The 8-bit byte and the ASCII code have become inseparable from microcomputing conventions, to the point that an attempt to work outside of these standards must necessarily consign a resulting system to limited use.

Even though part of the ASCII coding is taken up with control codes, and extended ASCII may be used for unusual pictorial symbols, there is ample room in the ASCII convention for the alphabetic symbols, superscripts, subscripts and punctuation of the Western languages that are alphabetic in nature, for which a simple character-coding reassignment is possible, albeit some complications necessarily arise, especially when variant forms of a symbol are found in different environments (as in Arabic, which has different written forms, depending upon whether the symbol appears alone, or at the beginning, end, or middle of a word, and which also obligatorily subscripts specific symbols and joins certain pairs of symbols in ligatures when they occur next to each other).

Ideographic languages, on the contrary, cannot fit into the limited symbol distinction capabilities of an 8-bit byte. The minimum number of symbols taught to Japanese school children in order that they may begin to read and write their language is 880; knowledge of more than 3,000 symbols is required to be able to read a Japanese newspaper, and a Japanese student graduates from high school with knowledge of about 4,000 Japanese characters. The

full Mandarin symbol system from which the 50,000 Japanese kanji are derived amounts to more than 65,000 symbols. Given this much greater need for symbol distinctions, it is natural and tempting to believe that the evolution to 16-bit and 32-bit central processing units (CPUS) should perhaps provide an environment in which more sequential bits can be used to encode a byte (16 bits allow 65,536 distinct symbols, and 24 bits allow 16,777,216). The kind of solution that has been devised for bilingual processors, however, does not require bytes longer that 8 bits nor does it use two (or more) bytes to encode every symbol; one reason for avoiding these arrangements is because it would be very wasteful of memory to require every symbol to be encoded in terms of an increased number of bits.

The preferred solution, rather, is to employ the algorithmic capability of the computer (and thus apply the greater computing capacity of the 16-bit and 32-bit CPUS) to recognize a combination of "shift alphabet" code (binary 11111111) and "language specifier" code (e.g., 00000000 for the roman alphabet, 00100110 for Greek, 00100111 for Russian, etc.), and correspondingly alter the interpretation of input and output symbols and ordering routines. Within this scheme, if the shift alphabet code (11111111) is followed by a "superalphabet shift" code (11111111), a third byte is now interpreted as a "superalphabet specifier," and each subsequent symbol is interpreted using 2 bytes at a time (for a total of 16 bits, allowing the distinction of 65,536 symbols). This procedure is quite adequate for managing ideographic symbol systems.

As mentioned by Joseph D. Becker, this "flexible encoding" strategy created by Gael Curry of Xerox's Office Systems Division allows the representation of more than 16 million characters while using memory very economically, because double-byte coding is used only when needed, as in the cases of Mandarin and Japanese. In Becker's words, "It allows text in any mixture of living languages to be represented economically in the computer as a sequence of bytes." This strategy also serendipitously maintains downward compatibility between the more powerful 16-bit and 32-bit CPUS and texts created on other machines, because the 8-bit basis for character encoding remains constant (and will also remain open to further increases in power, e.g., via a 64-bit CPU).

INPUT AND OUTPUT SOLUTIONS

For phonetic writing systems, the inputting of symbols is relatively straightforward. The combined sequence of the alphabet shift and the alphabet specifier codes (which are associated with a key or combination of keys on the monitor's keyboard) signals to the system that all following character inputs are to be interpreted according to the coding scheme devised for the language that has been specified. As a result, appropriate screen drivers and print drivers are activated, which display the specified language's symbols on the monitor screen as they are typed and allow those symbols to be presented via various kinds of printers. Whenever the alphabet shift code reappears followed by a new alphabet specifier, the coding scheme for the previous language is turned off and is replaced by the coding scheme for the newly specified language. This change in language specification may occur as frequently as required, anywhere in the text and, thus permits mixing the symbols from various languages with total flexibility, even within the same word.

The symbol scheme devised for a given language is referred to as that language's keyboard, and the operator has the option of calling to the monito

screen a miniature visual representation of the monitor's keyboard that has the specified language's symbols displayed on the keys to which they are assigned. This screen representation of a language's symbol coding scheme is called a "virtual keyboard" for that language. Every language included in a bilingual processor has its own virtual keyboard available, which may serve as a guide for typing the language's symbols by pressing the corresponding keys on the monitor's keyboard or which may be used more directly by employing a mouse to position an arrow on the symbol desired. This symbol is then input to the character stream by clicking a button on the mouse.

Inputting conventions require that symbols be input according to their phonetic sequence, that is, the sequence in which they are spoken, even though writing conventions may turn them around or display them in strangely complicated fashions (e.g., the native word for Hindi is written in the sequence *ihndi*). Symbols may also take different forms, depending upon where they occur in relation to other sounds, the beginning, middle, and end of words, etc. (e.g., the Greek *s* "sigma" has one form in the middle of a word and another at the end). Finally, the writing sequence may be from left to right or from right to left (assuming that Oriental script is converted to a horizontal format). Notwithstanding the complications found in the written representation of differing languages, the requirement of phonetic sequencing for symbol input gives the internally coded text a uniformity that makes it much easier to manage than one might judge based on the written representation.

The differences between phonetic internal coding and the arbitrary complications of written representation are mediated by algorithms idiosyncratic to each language that are made part of the screen and print drivers for the given language. Thus, if the code for the Greek sigma is found at the end of a word, it is written using its word–final form; otherwise, the normal form is used. If the Arabic *lam* and *alif* occur next to each other, they are combined in a ligature; and if the text is identified as Arabic, the output sequence is from right to left, even if the preceding and following text on the same line must be output from left to right.

The inputting procedures for ideographic writing systems involve a special degree of complication. Chinese and Japanese share similar characteristics in this regard, but Japanese, given its combination of kanji ideogram roots with phonetic kana grammatical endings, is the most complicated case of all. It is possible to input kanji symbols (or Mandarin characters) directly from a keyboard that contains hundreds of keys combined with other shift keys or, alternatively, to analyze the kanji according to some set of abstract characteristics and enter the coding for those characteristics, which is then converted into the proper kanji symbol. However, both of these methods are very inefficient in contrast to the preferred method, which is called "phonetic conversion." Phonetic conversion combines the notational efficiency of a phonetic symbol system with the computational capability of the computer to achieve a totally new level of efficiency.

In phonetic conversion, the typist inputs a phonetic spelling of a word (either via a native kana keyboard or via an English *romaji*, for roman, keyboard). The bilingual processor searches a lexical and grammatical database for possible interpretations of the word (which generally is composed of two-character combinations, allowing an early narrowing of the interpretation) and creates a decision tree that assigns degrees of probability to the various possible interpretations, based on other accompanying text. The most probable interpretations are displayed in the proper combination of kanji and kana

characters on a virtual keyboard, from which the typist then chooses the one that is appropriate by pressing or pointing to its assigned virtual key and thus adding its combination of characters to the character stream. With phonetic conversion, the average typist can touch-type Japanese at a rate of 50 characters per minute, and speed typists can attain rates of 150 character per minute. Similar procedures can be applied to Mandarin while using the *pinyin* roman alphabet.

Implied in all that has been said is the existence of screen and print drivers, which output the written language symbols using various character representation techniques. As part of the driver for each language, the language's symbols are defined as a pattern of dots for bit-mapped images, or as a configuration of combined Bezier curves for outline images—the dots print positively and compose the bit-mapped character, whereas the Bezier curves form a smooth outline that is blacked (or whited) in by a filling algorithm. Bit-mapped symbols are printed to the screen and also may form the basis for printing on dot–matrix and laserjet printers; outline symbols may be printed on laserprinters or typesetters but not to the screen (the system will generally choose a roughly equivalent bit-mapped screen font in relation to an outline laser printer font, or alternatively, the operator specifies the desired screen and laser printer fonts).

Bit-mapped symbols may appear ragged if the dot resolution is low in relation to the desired detail and, in some cases, may not even allow the proper representation of very intricate symbols; furthermore, bit-mapped symbols must be created separately for each different point size and style (e.g., italic or bold). Outline symbols are generally smooth but are also dependent on adequate dot resolution (near typeset-quality results can be gotten from the present laser-printer standard of 300 dots per inch); the equation-based Bezier curves that define outline images, moreover, can be algorithmically scaled up or down in size or modified for changes in style, thus allowing flexible point-size and style variations based on relatively limited font memory requirements.

The creation of bit-mapped or outline symbols may be done by virtually anyone, using software especially devised for that purpose (e.g., Altsys Corporation's Fontastic for bit-mapped symbols and Fontographer for outline symbols); however, the attainment of an aesthetically satisfying and readable result, especially when creating an entire font or symbol set, whose characters must appear in many different possible orientations and combinations, demands extensive knowledge of the centuries-old lore of typeface design, which is a precise, subtle, and complex discipline.

As is the case for printing the Western languages via electronic publishing systems, many different computer fonts that are based on existing typefaces may be purchased. A good first stop in searching for a standard font is a laser-printer vendor, who is necessarily in contact with a company that has access to considerable font resources. For foreign language fonts, especially regarding non-Western and less common languages, the search should extend to a desktop publishing journal (particularly the newly launched *Publish!*) and to the bulletin boards of the electronic publishing special interest groups (SIGs) associated with the various information services; these sources will generally lead to font designers at universities, who distribute their creations free of charge (credit to be given on the preliminary pages of a resulting publication is implied) or via share-ware arrangements. Alternatively, one can acquire the software and invest the time and love required to create one's own symbol fonts—the degree of technical and aesthetic satisfaction that may be found in this kind of enterprise can be

savored by contemplating the delicacy and unique personalities of the rhythmic dots and arches of Thai script, the flowing curves of Arabic writing, or the feathered edges of a Japanese character.

PREFERENCE FOR THE MICROCOMPUTING ENVIRONMENT

One of the apparent paradoxes of microcomputing is the dedication of microcomputing resources to a problem as complex as bilingual processing, when it might more reasonably be expected that this kind of question could benefit from the greater computing power available on a minicomputer or a mainframe and that the constituency of mainframe users should be sufficiently broad so as to create demands for bilingual capabilities. The fact is, however, that bilingual processing has been concentrated principally in the microcomputing environment. One reason for this development is the inertia associated with mainframe use, whose resources have been dedicated more to scientific and mathematical or business use. In both cases, any move away from the EBDIC or ASCII standard codes creates conflicts with other established parts of the computing system and is thus seen as a distraction from the scientific or business user's immediate purpose (which, to that user, is the only purpose for using the computer).

The multiuser capability of minis and mainframes, which could be expected to create a demand for bilingual processing capabilities, does the opposite: Because the majority of users are accustomed to the English-language basis for computation, including word-processing, and are more interested in their system acquiring upgrades for scientific, programming, or business purposes, rather than wandering into the thicket of conflicting codes that may be associated with bilingual processing, a natural common denominator exists in current multiuser environments, which gives a low budgeting priority to bilingual processing upgrades. Some idea of the extent to which the creation of foreign language capabilities can collide with a computing system is found in articles that discuss the adaptation of UNIX to European languages and to Japanese (*UNIX Review*, December 1985 and February 1987).

Paradoxically also, the current increasing concentration on broader and more harmonious networking capabilities tends to conflict with bilingual processing again because of the collision of networking control codes with the bilingual processor's encoding requirements, which adds one more bothersome factor to a network development environment already confused by a seemingly unmanageable set of competing requirements. I would analyze the conflicts mentioned in these three paragraphs in terms of underlying conflicts between the relative values assigned to computing capability, system salability, business profit-making support, and universalized international communication. In the current context, the value attached to universalized international communication is naturally lower than the values attached to the other factors mentioned; nevertheless, the increasing awareness of the growing competitiveness of foreign societies with regard to the generation of technology and its application to profit-making business should at some time bring these conflicting values into an alignment that will allow the implementation of bilingual processing on a large scale to drive adjustments in other parts of computing systems and applications.

Bilingual processing has found fertile ground in the microcomputing environment because personal computers and individual workstations can be tailored to specific requirements, in this case, the need to manage foreign language text. The development of the bilingual processor, moreover, has

coincided with the rapid increase in microcomputing power, coupled with rapidly decreasing prices. This situation has created for the language-interested user the ability to acquire the system that the user needs, which aligns with the microcomputer (and text processor) producers' requirement to sell their systems in large volume. Interestingly, as the microcomputing revolution advances, microcomputers are now approaching mainframe power, and it may be that via this expansion of the microcomputing domain true bilingual processing will become a standard part of internationally networked computationally based communication.

BIBLIOGRAPHY

Becker, Joseph D., "Multilingual Word Processing," Sci. Am., July 1984, pp. 96–107.

Fraenkel, Gerd, *Writing Systems*. Ginn and Company, Boston, May, 1965.

Gleason, H.A., Jr. *An Introduction to Descriptive Linguistics*. Holt, Rinehart, and Winston, New York, 1961, pp. 409–439.

Gleason, H.A., Jr. *Linguistics and English Grammar*. New York, Holt, Rinehart, and Winston, 1965, pp. 108–111; 397–418.

Hodgkin, Adam, "New Technologies in Printing and Publishing: The Present of the Written Word," in *The Written Word: Literacy in Transition* (Gerd Baumann, ed.), Clarendon Press, Oxford, England, 1986, pp. 151–169.

Nakanishi, Akira, *Writing Systems of the World*, Charles E. Tuttle, 1980.

Pfaffenberger, Bryan, *The Scholar's Personal Computing Handbook*, Little, Brown and Company, Boston, MA, 1986, pp. 52–67; 131–135.

Yamada, Hisao, "A Historical Study of Typewriters and Typing Methods: From the Position of Planning Japanese Parallels." *J. Inf. Process*. 2(4), 175–202 (February 1980).

BRUCE STIEHM

BIOS

INTRODUCTION

BIOS (Basic Input/Output System) is a collection of software routines that contain the instructions to operate a standard set of input and output devices connected to a computer system. The BIOS routines are normally written and copyrighted by the microcomputer system manufacturer and provided as ROM (read-only memory).

PURPOSE

In the early days of computing, an application program had direct access to the computer hardware and was completely responsible for input and output. Many computer systems provided the programmer with a single instruction that would read the contents of a punched card into memory and another instruction to output the contents of a block of memory as a line on the printer. The available input/output (I/O) devices operated sequentially and were relatively simple, with error detection causing the computer to stop rather than use error recovery code. The introduction of faster and more sophisticated computers with disk drives and communications devices and the utilization of high-level programming languages (e.g., FORTRAN) necessitated the provision of input/output subroutines and then encouraged the development of operating systems. The operating system assumed responsibility for processing a stream of jobs, and the Input/Output Control System (IOCS) performed any requested input or output, resulting in a significant improvement in programmer productivity and a reduction in operator intervention.

BIOS provides routines for accessing I/O devices, such as the keyboard, screen, disk drives, and printer, attached to a microcomputer. This involves the following activities:

- Checking that the device is ready
- Initiating the data transfer
- Checking for successful transfer
- Performing error correction procedures, if necessary
- Reporting successful completion or permanent error condition

BIOS is also invaluable for determining the equipment configuration and verifying the operation of the hardware on power-up (Power On Self Test, or POST).

The BIOS design of both Apple II and IBM personal computers (PCs) allows any user the ability to power on the machine and then proceed to enter and run a BASIC program using a version of BASIC, which is also available in ROM.

Most microcomputers are now equipped with disk drives and normally use a disk-based operating system (DOS). DOS routines supplement those in BIOS to provide extended services for using I/O devices. The division of responsibilities between BIOS and DOS for accessing disk storage devices is illustrated by the IBM PC:

- BIOS can read or write to a specified sector, track, and side of a diskette.
- DOS has responsibility for management of space on the diskette, including maintenance of a directory of the files and recording the space that has been allocated.
- DOS provides access to files and their records as logical entities and not by their physical location.

Application programs usually specify I/O activities to the operating system, with the operating system transforming the request into one or more calls to the BIOS routines. DOS is less device dependent than BIOS because redirection of I/O is available; for example, output normally destined for the screen can be redirected to a file.

The availability of BIOS and DOS for I/O activities simplifies programming, encourages standardization, and removes much configuration dependenc from application programs.

APPLE II AND APPLE II PLUS

The Apple II and Apple II Plus PCs are based upon the Synertek/MOS technology 6502 microprocessor. The Apple design allocated the top 16Kb of the available address range of 64Kb for ROM to contain programs and data that are available immediately after power-up. The contents of ROM vary, depending upon the Apple model and the peripherals used, but they always contain a monitor program and a version of BASIC. The system design was such that a minimum system had only 16Kb of RAM (random access memory) for user programs, a keyboard and a display screen, and the provision of an interface for a cassette tape recorder. Additional devices, such as a printer, serial communications port, and diskette drives, can be added by installing a plug-in adapter that normally contains additional ROM progamming to support the device.

The Apple II Plus has the Autostart ROM as the system monitor. When the computer is powered up it will check ROM address space to determine whether a Disk Controller Card is installed, and if found it will then bootstrap DOS from the diskette rather than starting execution of BASIC located in ROM.

The monitor ROM program (occupying only 2Kb) provides I/O routines for transferring data between memory and the screen, keyboard, or cassette recorder. Routines are also included to assist in writing and debugging programs, including the ability to display and alter memory and register contents.

APPLE MACINTOSH

The Apple Macintosh uses the Motorola 68000 processor, which provides a very large addressable memory. The minimum configuration includes screen, keyboard, mouse, disk drive, and printer interface. The Macintosh Plus hardware is closely coupled to a user-friendly operating environment based upon 128Kb of ROM-based software (the Macintosh II has 256Kb of ROM). In this type of environment, the separation of BIOS and operating system has all but disappeared as far as the application program is concerned.

IBM PC

The IBM PC, as announced in August 1981, is based upon the use of the Intel 8088 processor. The 8088 (and 8086) has 1-Mb memory address space, and the allocation of this address space is part of the design of the IBM PC. The design allows for the addition of new features (e.g., higher resolution displays) and new machines (80286 and 80386). A machine that is IBM PC compatible requires BIOS routines that provide the same services as those provided in the proprietary IBM ROM BIOS.

The processor starts execution of instructions at a fixed location at the high end of the memory address space, which is occupied by ROM BIOS. The ROM BIOS uses 8Kb in PC and PC/XT computers and provides power-on diagnostic services, together with basic routines (BIOS) for accessing devices such as the keyboard, screen, printer, and disk drives. After successful completion of hardware diagnostics and system initialization, BIOS attempts to locate and load an operating system at track 0, sector 1 of drive A. If the A drive has not been loaded with a diskette, it will execute BASIC, which is provided as additional ROM. The original IBM PC was available without disk drives, and this configuration did not have access to the more versatile and powerful routines provided by DOS but was able to utilize a cassette for file storage using BIOS routines. DOS makes substantial use of BIOS routines, but with other operating systems BIOS routines may be used only at power-on time and to permit the loading of the boot record of the operating system from disk.

Application programs that utilize DOS and BIOS interrupts for performing input and output retain compatibility with future releases of DOS and new devices. A program that writes directly to the video memory may not work satisfactorily with windowing environments such as TOPVIEW and may require modification to support new types of displays and display adapters. For example, INT 10H (10 hexadecimal) invokes BIOS to perform screen operations by causing an immediate branch to the memory location specified in locations 40H, 41H, 42H, and 43H (40H is obtained by multiplying the interrupt number, in this case 10H by 4). The design allows for interrupts 0H through 0FFH (0–255 decimal); consequently, memory locations below 400H are reserved for the interrupt vectors. Interrupts from 0H through 0FH are reserved for hardware-generated interrupts, and 10H through 1FH are used for calling BIOS routines. An application program that jumps directly to an address in BIOS will probably not operate successfully on a machine with a different version of BIOS.

There are many different activities for screen I/O, and the desired service is obtained by loading registers to specify the required service prior to issuing the INT 10H instruction. For example, there is a scroll-up routine that will move a designated block of characters on the screen a specified number of rows. The routine will determine the type of screen display used (monochrome, color graphics, etc.) and execute appropriate instructions.

There are two BIOS routines for the keyboard. A hardware interrupt is generated by the keyboard each time that a key is depressed, released, or held down for a certain interval (typematic). This immediately invokes a BIOS routine that assembles extended ASCII characters into a keyboard buffer. A beep is sounded if the buffer becomes full to advise that the keystroke has not been accepted. This routine also has responsibility for recognizing certain reserved key combinations, such as SHIFT-PRSC and CTL-ALT-DEL and initiating appropriate action. The second routine is invoked by a software interrupt when a program requires the first character in the buffer or wishes to check whether or not the buffer is empty.

The introduction of memory resident utilities, such as Borland's Sidekick, requires modification to be made to the standard BIOS keyboard routines because it is necessary to recognize previously unused key combina tions such as CTL-ALT. This can be done by changing the interrupt vector for INT 09H to point to a routine in the memory resident utility that can check for a specified keystroke combination before returning to the standard keyboard interrupt routine.

The original BIOS was upgraded so that additional BIOS routines can be provided on device adapter cards to supplement the standard services. For example, the additional BIOS for fixed disk operations on the PC/XT is physically located on the fixed disk adapter card, because fixed disks were not part of the original IBM offering. To boot DOS from the hard disk instead of the A drive, it was necessary to provide BIOS code to check the A drive for a diskette before loading a boot record from the fixed disk. At power-up, the execution of instructions in the primary BIOS checks for additional ROM BIOS code by scanning each 2K block of memory in the permissible ROM range (C0000H through F4000H) for a block that starts with a specified pattern of bytes (Hex 55AA). As each addition ROM is identified, it will be used to perform diagnostics and any required initialization, such as changing the interrupt vectors, prior to returning control to the primary BIOS. BIOS has also been incorporated into network adapters (NETBIOS) and enhanced graphics adapters (EGA). For example, when the EGA is used, the software interrupt, INT 10H, will access routines on the EGA ROM because the interrupt vector has been changed to point to the EGA ROM.

The BIOS interface for accessing disk drives provides relatively low-level support because it requires the calling program to specify the drive, head, track, and sector to be used. Application programs will normally utilize DOS services, because DOS takes responsibility for the allocation of disk space and allows the program to operate with logical entities of records and files.

The evolution of operating systems for the IBM PC, including the 1987 announcement of an enhanced operating system (OS/2), will reduce the direct calling of BIOS routines by application programs. In a multitasking environment, an application program cannot be permitted to change interrup vectors, or access the screen or other devices directly. Standard application programming interfaces are now available to ensure that a newly developed application program will run under both DOS and OS/2.

Additional information on the BIOS for the IBM PC and PC/AT, including program listings, is published by IBM as technical reference manuals.

FUTURE DEVELOPMENTS

New processors with very large addressable memories, coupled with the continually decreasing cost of both ROM and RAM, allow the system designer much greater freedom to develop large, powerful, and user-friendly operating systems for the microcomputer. BIOS remains a fundamental and basic component of the operating environment but will seldom be accessed directly by application programs in the future, except where unorthodox use of the keyboard, fast video action, and copy protection are very important.

HUGH R. HOWSON
W. DAVID THORPE

BURROUGHS CORPORATION
Burroughs History

INTRODUCTION

During the last quarter of the 19th century, accountants were forced to keep pace with ever-increasing mechanization and a period of steady business expansion. Rapid growth in railroads, steel, mining, textiles, retailing, farm machinery, and other industries changed the nature of local partnership enterprises and signaled the emergence of the corporation. Goods and mail were moving swiftly on transcontinental railroad systems. The typewriter and telephone came into the modern office, speeding daily transactions. Meanwhile, clerks struggled to keep the accounts of burgeoning commerce with traditional bookkeeping methods.

Throughout history, mathematicians and bookkeepers had dreamed of devices that would automatically add columns of figures. Now, more than ever, these clerks saw the need for an adding machine. William Seward Burroughs was such a clerk. But, with his dream, he harbored determination and mechanical genius.

In the small town of Auburn, New York, young William Seward Burroughs worked as a clerk in the counting room of the Cayuga County National Bank. He found the drudgery of the work demanding and monotonous.

By the time he was 24 years old, the stress of his profession and the onset of tuberculosis began to take their toll on his health. Doctors advised Burroughs to choose a milder climate and a less taxing occupation. In 1882, Burroughs moved to St. Louis, Missouri, determined to produce the world's first practical adding and listing machine. A sympathetic machine shop owner, Joseph Boyer, encouraged Burroughs' work by giving him bench space at the Boyer Machine Shop, providing him with a young assistant (Alfred Doughty, later president of the company) and by supporting him financially from time to time.

Boyer later recalled, "There was Burroughs with his great idea, greater than any of us could fully appreciate, and with his meager capital of $300. Long before the first model was actually begun, his money was gone. But as his resources dwindled, his courage rose. I used to leave him at his bench in the evening and find him still there in the morning...

"When damp weather expanded the paper on which he worked, he resorted to polished sheets of copper, cutting his lines with the point of a needle. When he located the center, he did so under the microscope. When the polished copper proved tiresome to his eyes, he drew on polished zinc, chemically blackened, the lines showing white against the background of black. It was his way of drafting plans for what he knew must be minutely accurate."

Burroughs fashioned a working prototype of his adding machine in 1884 and filed patent application papers for the device early in 1885. Thus, at the age of 28, he established himself as the inventor of the world's first practical adding and listing machine. Although Burroughs was not the first to create a calculating device, he was the first to engineer a machine capable of printing out a list of figures and automatically adding them together.

1880–1900

Burroughs turned to local businessmen who had expressed interest in his invention to form the American Arithmometer Company for the manufacture and marketing of his machine. At their first meeting on January 20, 1886, Thomas Metcalfe was elected president; Burroughs, vice president; Richard M. Scruggs, treasurer; and Metcalfe's older brother, William, although not a stockholder, secretary. This initial meeting and many subsequent meetings of the new company were held in the office of the Scruggs, Vandervoort, and Barney Dry Goods Company in St. Louis.

The first machines produced by the company reached the market in 1889. Only a few of these models were sold. Customers discovered that the adding machines were not entirely reliable; the handle had to be pulled "just so" or sums were inaccurate. Burroughs explained that the operators were not using the machine correctly, but efforts to educate users in the proper operation of the device were futile.

With the fledgling company facing possible bankruptcy, Burroughs went to work to find a solution. He locked himself in his shop and worked tirelessly until he emerged with the answer: a "dash pot" composed of a cylinder partially filled with oil, which automatically regulated the calculating mechanism of the machine.

Production of the revamped machines began in late 1891 at the Boyer Machine Shop, and before long, the first of the new machines was distributed.

In its first 10 years, the American Arithmometer Company grew to include a factory and office staff of 65 employees. In about 1895, the company divided the United States into three sales territories and hired three salesmen who often traveled thousands of miles to make one sale. About this time, another young salesman took on the entire Dominion of Canada as his sales territory.

In 1895, Sir William Joseph Cannon of Nottingham, England, acquired the rights to manufacture and market Burroughs' adding machines in all the countries of the Eastern Hemisphere. Thus, the Burroughs Adding and Registering Company Limited of Nottingham, England, was established as an independent corporation. In 1898, the Nottingham factory began manufacturing machines designed to make calculations for English currency.

Burroughs made several improvements in the original device, including a wider carriage and the ability to make duplicate copies on the paper roll. He also invented an automatic ribbon reverse, which later became standard for typewriters. Banks and other industries were being quickly converted to the usefulness and dependability of the adding machine, and by 1898, 1,000 machines had been sold.

Due to failing health, Burroughs retired from active participation in the company in 1893, although he continued his inventions work for the firm. Shortly afterwards, he moved to the quiet town of Citronelle, Alabama. On September 14, 1898, Burroughs died of tuberculosis. He was 41 years old.

1900–1919

Joseph E. Boyer, who had encouraged and supported the efforts of Burroughs for many years as owner of the Boyer Machine Company, became president of the American Arithmometer Company in 1902.

By this time, the company had outgrown its small St. Louis plant. Rather than expand its operations in St. Louis, company officials opted to move the firm north to Detroit, Michigan. Boyer, who had relocated his machine company to Detroit in 1900, worked with the company and Detroit officials in selecting the site for a new factory—a location near the northern limits of the city. Construction of the new plant, designed by Albert Kahn, began in the summer of 1904.

At dawn on October 8 of the same year, the company-chartered "Clover Leaf Limited" pulled out of the St. Louis depot. Aboard was an unusual assemblage of foremen and factory hands, wives and children—465 people in all—gramophones, rolltop desks, crockery, and adding machines. The American Arithmometer Company was moving to Detroit.

After the long weekend trip, employees came to work on Monday to help unload the train and begin operations at the new factory. It wasn't long before adding machines were being produced again—71 in October, 283 in November, and 316 in December.

The American Arithmometer Company was renamed the Burroughs Adding Machine Company in 1905 in tribute to the man whose vision had led to its founding less than 20 years earlier.

That same year, employment rose to 1,200, and 7,800 machines were sold—as many as William Seward Burroughs had estimated the entire U.S. market would bear. The assets of the company were now valued at $5 million.

In the early 1900s, the company initiated a new department, boldly called the Department of Inventions. It was not misnamed, for the adaptations, improvements, and new products came in steady succession. Boyer, who himself had invented the pneumatic hammer and railway speed recorder took a particular interest in the work of this department.

An apprenticeship school, which was established in 1904 by the factory manager, Alfred Doughty, trained many young people in the mechanical trades. Another innovative department of the period, the Business Systems Department, was established to develop, collect, and publish information on new techniques for utilizing adding machines in a wide range of tasks and businesses.

The young company was also expanding by acquiring other businesses—particularly its competitors. In 1908, Burroughs purchased the assets of the Burroughs Adding and Registering Company Limited of Nottingham, England, which included the Eastern Hemisphere rights that had been sold to William Joseph Cannon in 1895. The following year, the company success-

fully moved to acquire the Universal Adding Machine Company of St. Louis and the Pike Adding Machine Company of Orange, New Jersey.

The company product line continued to expand during this period. In 1909, Burroughs introduced the duplex adding machine, featuring both totals and subtotals. A nonlisting calculator designed specifically for banking needs was produced in 1911. That same year, the company began to market the subtracting-adding machine. Its capacity to add, subtract, and list figures made it the industry's first practical bookkeeping machine.

As the company developed new and better products, it also enhanced its marketing efforts with an international sales force. Sales agents carried Burroughs products to every corner of the business world. But it wasn't enough just to sell the machine; the devices also had to be maintained and serviced.

In the company's earliest days, machines had been repaired by local contractors--often the town blacksmith, watchmaker, or bicycle shop owner. As the use of Burroughs machines became widespread, the company organized a formal service department with a staff of representatives responsible for repair and routine inspection of Burroughs products.

World War I saw many Burroughs men leave for service. The company supported them with letters and publications from home. One serviceman wrote to his Burroughs colleagues in 1918, "There is the same sort of spirit in this army as there is in the Burroughs organization—not a desire, but a determination to hit the mark, no matter how high it is set."

In an early demonstration of good corporate citizenship, the Burroughs organization contributed nearly $3 million to the U.S. government's Liberty Loan programs to support the nation's war efforts.

1920s

By year-end 1920, total employment in the factory and offices had risen to 8,500, and there were 3,500 people employed by sales agencies in the United States and Canada. Production was expanded by the establishment of a factory in Windsor, Ontario, opposite Detroit. Net earnings for 1923 were $4.4 million on sales of $28.4 million.

In May 1921, the company purchased the financially ailing Moon-Hopkins Billing Machine Company. Moon-Hopkins manufactured a unique adding, typing, and multiplying machine that was used primarily for transit listing in banks. Burroughs engineers improved and refined the product, and it became an important addition to the Burroughs product line.

Armed with an ever-increasing array of products, many of which could be tailored to meet the specific needs of customers, Burroughs salesmen were now able to penetrate markets previously closed to them. To prepare its sales force to meet this new challenge, the company required that all sales representatives be trained in the specialized selling techniques necessary to conduct Burroughs business. After 1921, Burroughs only hired salesmen having a college degree or at least some education above the high school level.

So rapidly was Burroughs' business growing that in January 1926, just 40 years after the original patent was awarded, managers and factory hands celebrated the production of the company's one millionth machine.

1930s

As the Depression chilled the national business climate, Burroughs entered the 1930s with a private sorrow. Joseph Boyer, the company's "grand old man," died of pneumonia on October 24, 1930, at the age of 82. Boyer had been a powerful force in the company's rapid development. Partially out of respect for Boyer's memory, his position as chairman of the board went unfilled for 16 years after his death.

Although the company continued to make money throughout the Depression, profits declined sharply, especially from 1930 through 1933.

Late in 1934, an article in the *Wall Street Journal* proclaimed, "Burroughs dominates the accounting machine field. Its line of products now comprises 450 standard models which, with variations within the line, run the total number of features to 2,000." Among the machines being produced were typewriters, cash registers, billing machines, adding machines, and accounting machines.

In 1937, Burroughs' net earnings surpassed $8 million. Optimistic about the future, the company began construction of a new plant in Plymouth, Michigan. A five-story factory and power house were completed in 1938 to relieve crowded conditions at the Detroit main plant and expand production capabilities. The company moved its printing division, ribbons department, direct mail, chair and stand assembly, chair manufacturing, and finishing units to Plymouth.

1940s

During World War II, the company cooperated in the national defense program by restricting the production of Burroughs machines to the needs of the armed forces and other war contractors.

In 1944, Burroughs was awarded an Army-Navy "E" pennant for outstanding achievement in the production of war material, principally the Norden bombsight. The bombsight made accurate, high-altitude bombing possible and was considered by some military authorities as a significant factor in shortening the war. Previously, mass production of the bombsight, a very advanced precision instrument, was believed impossible due to the half-millionth of an inch tolerance required. Although the company was not the sole supplier of the bombsight, it built the largest number of units by far.

By the end of World War II, there were 558 sales branches and service centers in the United States and Canada and 215 branches and dealer offices overseas. Company sales had grown to $46 million. John S. Coleman, who had acted as a company liaison with Washington authorities early in the war, was named president of the Company in 1946.

A pent-up need for commercial business machines spurred increased postwar production. In 1949, after nearly 15 years of development, Burroughs introduced the versatile Sensimatic accounting machine, which could perform dozens of accounting and bookkeeping tasks mechanically, switching from one to another at the turn of a selector knob. Later models, marketed as Sensitronics, utilized electromechanics magnetics and electronic circuitry to increase the speed and efficiency of the operator. These products became the standard of bookkeeping automation throughout the 1950s.

Although Burroughs continued its leadership in the production of commercial business machines in the prosperous postwar period, the era signaled a major turning point for the company. The war had accelerated the early development of electronic technology, including the potential for electronic data processing. Under Coleman's leadership, the decision was made to begin a full program of electronic research and development.

Dr. Irven Travis, who had worked on the Army's ENIAC project while at the University of Pennsylvania's Moore School of Electrical Engineering, was appointed director of research in 1949. The company's newly established research center was first located in rented facilities in Philadelphia, Pennsylvania, in 1949.

1950s

The nature of the industry and the company were changing swiftly and dramatically by the 1950s. Reflecting the company's new and diverse operations, the Burroughs Adding Machine Company was renamed the Burroughs Corporation in 1953.

In 1952, an Electronic Instrument Division was set up in Philadelphia to manufacture and market scientific instruments and electronic memory components and systems; and, in 1954, the company established permanent residence and development facilities in Paoli, Pennsylvania.

Early in the decade, Burroughs initiated experiments that were aimed at developing a series of computers specifically for business problem-solving. In 1954, Burroughs introduced the E 101, a desk-size electronic digital computer for scientific and engineering applications. Later, Series E systems, such as the E 2000, and counterpart Series F systems became widely accepted and, for many years, were Burroughs' leading products for accounting applications in business, industry, and banking.

In parallel with Burroughs' development of electronic products for accounting applications, the company expanded its capability for development of larger, multipurpose computer systems. In 1952, Burroughs built an enhanced memory system for ENIAC, one of the world's first electronic computers. The Burroughs system expanded ENIAC's memory sixfold and provided high-speed access to the memory, increasing the rate of computation.

The first digital electronic computer built by the company was the Burroughs Laboratory Computer, installed in 1951 at the research center in Philadelphia. Two years later, Burroughs engineers delivered an enhanced version of this system, known as the Unitized Digital Electronic Computer (UDEC), to Wayne University in Detroit. Both were general purpose computers constructed of interconnecting series of "pulse-control units," similar to those used in the Whirlwind I Computer, developed at MIT in 1947.

Several key acquisitions during the decade further strengthened the company's electronic development program. The company acquired Control Instrument Company, a designer and manufacturer of electronic instruments and fire control devices for the U.S. Navy, in 1951. In 1954, Burroughs purchased Haydu Brothers, a manufacturer of vacuum tubes and other electronic components, which enabled the company to make rapid strides in the then incipient field of electronic display technology.

Burroughs acquired the ElectroData Corporation in 1956 to significantly expand the company's base in the computer industry. ElectroData, at the time, the world's third ranking computer manufacturer, provided Burroughs with superior engineering and manufacturing capacity. That same year, Burroughs' Great Valley Laboratories, an engineering research complex, opened near Paoli.

Burroughs' development of a full range of computer systems progressed steadily. The ElectroData Division introduced the Datatron 220 in the fall of 1957, the industry's first medium-priced digital system with core memory. The B 251 Visible Record Computer for banking applications was introduced 2 years later. The Burroughs B 201 high-speed document sorter, developed for use in the banking industry and adapted to the emerging magnetic ink character recognition (MICR) technology, was introduced in the late 1950s. The company's capabilities in the graphic arts enabled it to take the lead in the introduction of MICR systems in banks, first in the United States, then in Canada, and later overseas.

One of the most exciting and significant advancements of the decade came in 1957 when Burroughs installed the world's first operational transistorized computer at Cape Canaveral, Florida. This was the Ground Guidance Computer, used in guiding the launch at Atlas intercontinental ballistic missiles. Subsequent versions were designed for use in the Mercury and Gemini manned space flights.

By the late 1950s, the company was becoming a major force in the emerging electronic data processing industry. Several factors contributed to this success. Coleman's commitment to the future of electronics and his allocation of significant company resources for research and development set the foundation for growth. Acquisition of the Control Instrument Company and the ElectroData Corporation stimulated advanced electronics work at Burroughs.

Having earned a reputation for electronics expertise, Burroughs was able to attract many government contracts which, in turn, fueled the company's research efforts with greater financial resources and expanded research opportunities.

In 1954, Burroughs entered the Semi-Automatic Ground Environment (SAGE) program, a continental air defense system, and supplied the AN/FST-2 radar data processor, which was the chief building block for the SAGE radar and data processing network. Burroughs research and development in the SAGE program established many important procedures in the theory and techniques of basic radar data processing. In 1959, Burroughs began development of the U.S. Air Force Airborne Long-Range Input (ALRI) system, a seaward extension of SAGE and was assigned the responsibilities of prime contractor and systems manager. The Burroughs Stabilization Data Computer (SDC) was the nerve center of the Polaris missile-equipped submarines, the first of which was launched by the U.S. Navy in June 1959.

With the acquisition of several office supplies companies, Burroughs continued to expand its operations in office consumables. Burroughs had purchased Mittag and Volger, Inc., and Acme Carbon and Ribbon Company, Ltd., manufacturers of carbon paper and machine ribbons, in 1949. The acquisition in 1955 of the Todd Company, a major supplier of checks, business forms, and check protectors further enhanced Burroughs product capability in the office supplies market.

The company was becoming a single source supplier of a wide variety of products for business and information management. By 1959, total em-

ployment had grown to 35,000 and revenue to almost $360 million. The company maintained sales and service agencies in 66 countries.

1960s

By the 1960s, computers were finding their way into the mainstream of business, science, and government.

Burroughs began marketing the B 200 series of small- to medium-scale solid-state computers in 1961. Datavan, a unique marketing program originated in 1963, demonstrated the B 200 data processing systems to customers and prospective customers in major U.S. cities. A complete B 280 system was transported by van on an 8-month, 12,000-mile transcontinental tour, reaching 25,000 people in 22 U.S. cities.

The Burroughs B 5000 solid-state modular test data processing system was introduced in 1961. Designed to allow hardware and software to work to the user's maximum benefit, it featured such pioneering capabilities as automatic multiprogramming and multiprocessing, exclusive use of higher level languages, and virtual memory. The B 5000 was considered by many industry analysts as being a decade ahead of its time.

It was at this juncture that Burroughs made the design decision to make hardware and software work together to the user's maximum advantage, assuring that hardware and software development teams worked in concert to produce a computer architecture that would deliver the performance, productivity, ease of use, and flexibility that would solve the customer's information problems and be cost effective.

The B 5000 was followed by the enhanced B 5500 system in 1964. Within the "500" family, and within all subsequent generations of Burroughs computer systems, users could move from an entry-level machine to the top of the line and from generation to generation without the expense and delay of reprogramming.

The 500 family served a broad cross section of data processing requirements in fields such as banking, manufacturing, and government. It solidified Burroughs' position in the computer industry and was a key element in Burroughs' emergence as a world class information systems company.

In 1961, Burroughs was named by the U.S. Air Force as hardware contractor for the North American Air Defense (NORAD) Command's 425L program, a combat operations computer complex and data display system. The system was used to make split-second evaluations of threats to the North American continent using input from satellites and radar around the world.

Serving as prime contractor to the U.S. Air Force Systems Command's Electronic Systems Division in the early 1960s, Burroughs designed and produced the D 825 systems for the Back-Up Interceptor Control (BUIC) program, a continental air defense system. The Burroughs D 825 data processing system represented a new approach to computer organization and the first totally modular computer system.

Burroughs' success at solving data processing problems took another evolutionary step in the late 1960s with the introduction of the Series TC terminal computers and the Series L minicomputers. The Series TC internally programmed computers were designed for use with on-line data processing systems and could function as either terminals or independent computers. The Series L was designed primarily as a self-sufficient billing computer but featured a data communications option that enabled it to operate on-line as a terminal computer. Both Series TC and Series L minicomputers were well received by all types of customers, and more than 140,000 units were sold worldwide in 10 years.

In 1966, under the leadership of Chairman Ray R. Eppert, Burroughs reached another landmark—becoming a half-billion dollar company. Eppert retired in 1966 and Ray W. Macdonald was elected as the new chairman.

In the late 1960s, Burroughs began the first phase of a major, long-range capital expenditure program for the expansion of its worldwide production resources. Six new facilities were opened in 1967, including three in the United States and one each in Belgium, Brazil, and Maxico.

1970s

High tech was the catch phrase of the 1970s as computers became an integral part of modern life. Burroughs, now an established leader in the computer industry, approached the decade on strong financial and technological footing.

A new five-story, 600,000-square-foot World Headquarters building was opened in Detroit in 1971 on land the company had owned since 1904. In 1972, the company's revenue topped $1 billion.

The company used its growing resources to develop several complete new families of computer systems—from minicomputers to large-scale computers—and to support them with a full range of related software products, computer peripherals, términals and data communications systems, and data processing equipment.

Throughout the 1970s, Burroughs introduced a number of computer families, each compatible with its predecessor; following the 500 family, Burroughs developed the increasingly more compact "700," "800," and "900" families in the 1970s. Each new series incorporated advanced technology for greater performance and throughput. These computer systems were broad in scale and application, encompassing small entry-level systems, medium-scale systems, and large-scale mainframes.

In 1972, Burroughs installed ILLIAC IV, the world's largest and most powerful computer at the National Aeronautics and Space Administration's (NASA) Ames Research Center in Mountain View, California. Burroughs was responsible for the design, development, and manufacture of ILLIAC IV. The system was conceived at the University of Illinois under the direction and funding of the Department of Defense's Advanced Research Projects Agency.

Burroughs was selected by the Society for Worldwide Interbank Financial Telecommunications (S.W.I.F.T.) in 1974 to supply data processing and data communication equipment for a new international telecommunications network linking 239 member banks. By 1985, S.W.I.F.T. had become the world's largest and most complex interbanking system, using Burroughs equipment to process 600,000 transactions a day among 1,200 banks in 42 countries.

As Burroughs products became more and more sophisticated, user education became an important aspect of the company's customer support program. By 1978, Burroughs offered a curriculum of more than 200 courses

in 55 training centers throughout the world, instructing customers in the use of Burroughs hardware, system software, and application program products.

Aware of a growing need for technical personnel, Burroughs significantly expanded its employee cooperative education and scholarship assistance programs in the 1970s. This was part of its long-term commitment to preparing more men and women for the highly specialized work of the data processing industry.

Ray W. Macdonald retired as chairman at the end of 1977, and Paul S. Mirabito took over as chairman of the board, president, and chief executive officer.

During the 1970s, the company also continued its developments in other areas of data processing by introducing products for data preparation and document handling; a full range of displays, keyboards, printing terminals, and related data communications computer systems; memory subsystems; high-speed printers; and software and special-purpose equipment for applications in principal lines of business, including banking and finance, manufacturing and distribution, government, health care, and education.

The expansion in data processing was paralleled by Burroughs' entry into the office automation market. The company entered the fascimile communications market in 1975 by acquiring Graphic Sciences, Inc., which produced equipment under the "dex" trademark, and entered the word processing market 1 year later by acquiring Redactron Corporation. The acquisition of the assets of Context Corporation in 1979 added an optical character recognition page-reader system to the growing range of office automation products.

While the Burroughs product line expanded, so too did employment and revenue. By the end of the decade, revenue approached $3 billion and employment exceeded 50,000.

1980s

Burroughs entered an area of fundamental change in the 1980s, directed by a new management team and driven by its commitment to supplying total information systems to specific market areas. Under the guidance of former U.S. Secretary of the Treasury, W. Michael Blumenthal, who became chairman of the board in 1980, and Dr. Paul G. Stern, named president in 1982, Burroughs sought to fill technology voids and strengthen areas targeted for growth.

In 1980, System Development Corporation, a leading information systems supplier for government agencies, was acquired. Its operations were consolidated in 1982 with those of Burroughs' Federal and Special Systems Group to form System Development Corporation—a Burroughs company. SDC now serves the U.S. government in fields from airspace management to command and intelligence systems and is a recognized leader in secure network and systems integration.

The 1981 acquisition of Memorex Corporation brought to Burroughs first-rate capabilities in computer storage devices, one of the fastest growing areas in the information systems industry. Today, Memorex is a complete plug-compatible peripherals supplier, providing storage devices for

Burroughs and other vendors' systems, complemented by a complete range of terminals, printers, and controllers.

Burroughs acquired Graphics Technology Corporation (Graftek) to strengthen the company's position in the important field of computer-aided design and manufacturing. In 1982, Burroughs introduced the B 20 family of powerful microcomputers developed by Convergent Technologies and built to Burroughs' specifications. The B 20s operate as stand-alone intelligent workstations or as components in distributed processing networks.

Much of the product development in the 1980s has been guided by the market's emphasis on fully integrated distributed processing networks. In 1984, Burroughs launched a new era in mainframe systems with its A Series of large-scale computer systems; the V Series of mainframes followed early in 1985. The A Series and V Series are the successors to the successful 900 family of computers and continue the company's commitment to cost-effective design principles and tradition of system compatibility.

The most recent entry into the A Series, the large-scale A 15, makes it possible for entry-level A 3 users to increase computer capacity up to 70 times without reprogramming. The V Series of medium-scale computer systems represents five generations of compatibility since the introduction of our 2500 and 3500 systems more than 20 years ago and features greater capacity and overall throughput than Burroughs' current systems.

Great strides have also been made during the decade in designing innovative system software. Burroughs' advanced program-generating software, LINC (Logic and Information Network Compiler), introduced in 1982, and LINC II, announced in 1985, can increase the productivity of application programmers up to 10 times and more. The new Interpro software, announced in 1985, brings the programming and operational ease of personal computing to Burroughs' largest systems.

The B 25 intelligent workstations and the XE 500 series of processors, both introduced in 1984, allow for modular expansion of distributed processing systems. By utilizing Burroughs' Office Management System II (OMS II) software, introduced in 1984, departmental clusters of B 25s or XE 500s can be connected to Burroughs large, medium, and small processors.

Refinements in Burroughs' 1981 office information systems (OFIS 1) led to another major advancement in the company's office automation strategy. The OFIS writer 25 (OW 25), introduced in early 1985, is a word processing system that utilized B 25 microcomputers with a specially designed keyboard and the OMS II word processing software. This is an important step toward Burroughs' goal of completely integrating office information—whether in the form of data, text, graphics, voice, or image.

Burroughs is now exploring advanced applicative programming languages and a radically different machine architecture; together, they promise to revolutionize the way people work with computers. These fifth-generation systems will feature concurrent processing and artificial intelligence, the ability to work in terms of logical relations that mimic the way people think.

The company's continuing dedication to quality has stimulated management innovations throughout the decade. In 1982, Burroughs established a corporate quality organization, charged with defining, measuring, and evaluating quality in every area of operations. The Corporate Quality Policy states, "We shall strive for excellence in all endeavors. We shall

set our goals to achieve total customer satisfaction and to deliver error-free, competitive products on time, with service second to none."

Early in the decade, the company implemented a worldwide system of customer service centers. These centers speed service response to customer calls, providing users with rapid and comprehensive service for all their needs. Toll-free "hot lines" have also been established to answer customers' questions regarding system software and application programs.

Along with this renewed commitment to quality and service, Burroughs initiated two new recognition programs for its worldwide force of nearly 65,000 employees in 100 countries. The Employee Suggestion Program, established in 1981, offers cash awards to employees in return for profitable ideas. In early 1983, Burroughs launched its Achievement Awards Plan to recognize employees for outstanding performance in all areas. Advancements were made in employee training programs with the construction in the early 1980s of major training facilities in Lisle, Illinois, in the United States and in Milton Keynes, England.

The company marked its 100th operating year in 1985. At the annual meeting of stockholders on April 3 of that year, Burroughs launched its centennial with the announcement of a new corporate identity program. The identity program grew out of an intensive, year-long study of the company, its employees and public, and a comprehensive audit of Burroughs' communications.

The major elements of the corporate identity program were a bold new symbol: new logotype, corporate colors of crimson and gray, and a strong positioning statement. This new identity became the foundation for all Burroughs communications, projecting a distinctive and focused image in the marketplace.

In his address at the annual meeting of 1985, Chairman W. Michael Blumenthal described the company at the turn of the century.

"As we enter our second century of working to meet the needs of our customers, we will continue to be innovative and imaginative. Our basic stance will not be one of closing out a century of Burroughs history but of launching another hundred years of progress.

"Our goals will remain the same—to achieve total customer satisfaction, to deliver error-free, competitive products on time, and to provide service second to none.

"We will continue our strategic commitment to targeted lines of business and to maintaining leadership in our own areas of excellence."

In June 1986, Burroughs and Sperry Corporation agreed to a merger in a transaction valued at over $4.4 billion. The new $10 billion company, yet unnamed at press time, is the second largest information systems company in the world.

BURROUGHS CORPORATION

CAD/CAM, INTEGRATING WITH CIM: THE MANAGEMENT CHALLENGE

For 20 years management from the president on down has been barraged with hundreds of acronyms, such as MIS, CAD, MRP, MRPII, CAM, CAE, and CIM. Some of us look back wistfully to the time when "getting out the iron" meant a simple phone call to the best expeditor. Why have we become so complex in our actions—our thinking—our planning? The fact is that complexity has increased because *integration* of our knowledge in the form of data has not occurred but remains fragmented. The organization of our people also remains fragmented and reflects the same condition. Although it is easy to make this statement, its implication and the importance of charge are not as obvious. Let us divide the problem and the challenge into five sections:

1. What is integration?
2. The impact on organization
   - A. Structure
   - B. Emotions
   - C. Knowledge
   - D. Data and flow
3. Old geezers—young tigers
4. A new look at old technologies
5. The future

WHAT IS INTEGRATION?

There was a time when one could request information about a part number from a master mechanic or production control supervisor and be told what it was, its material, where it was stored, how to machine it, the black book dimensions used to "adjust" the engineering design, and more. Through the force of memory, long employment, and tenacity, people made the manual system operate. In a way these people were an integrated system, for they had stored and remembered a host of diverse data generated by several different departments.

A few years ago, a famous pump manufacturer discovered that the assembly floor could no longer assemble pumps. It took a good deal of soul searching to discover that their skilled assemblers who had the know-how had retired. No written instructions existed to convert engineering design to a practical product.

Products are no longer as simple as Henry Ford's black Model-T. We have added hundreds of varieties, electronic controls, closer tolerances, NC tool manufacturing, multiplant operations, and many more variables to the equation.

Material requirements planning (MRP) and its associated data requirements and integration of knowledge began to integrate manufacturing and indicate not only the amount of data needed for the computer system to operate correctly but the disciplines required for an effective result. However, this "downstream" activity almost completely ignored the Engineering Department whose personnel were considered to design "things," with no direct involvement in the flow of data. Thus, many portions of engineering documentation were mechanized by the Data Processing Department or the Manufacturing Department without engineering involvement.

Introduction of CAD, CAE, and CAD/CAM have illustrated how a preponderance of data used by all departments is initiated in the Engineering Department. Thus, we have discovered that they must join the team and help us to integrate data and the corresponding flow of information.

Sharing data, not reinitiating it in each department, is integration.

The need to share data has caused a severe strain on the present organization.

THE IMPACT ON ORGANIZATION

Structure

In times past the sheer size of the manufacturing job required segmentation of authority and work effort. Engineering and Manufacturing organized themselves "vertically." Unfortunately, systems are organized horizontally between and through departments. Because each department is responsible for the accuracy of its own data and actions, one is naturally wary of using information generated by another. Figure 1 shows how vertical and horizontal relationships created a troublesome paradox.

Preparation of data for computer systems was initiated with great fanfare 20 or more years ago. Unfortunately, the electromechanical devices and small-scale computers available at the time combined with segmented functions created a series of computer systems that often were not compatible and had much redundant data and overlap of function. Any gesture to combine or merge these systems met with significant resistance, for such combinations implied relinquishing responsibility and authority, as well as dependence upon others for performance of their jobs.

Recently Steve Rosenthal of Boston University queried 64 experts in the field of factory automation. This group was almost unanimous in concluding that the most difficult problems in achieving computer-integrated manufacturing (CIM) are primarily managerial rather than technical. Chuck State of Wang argues that the Manufacturing Engineering Director cannot implement CIM; it is a matter for the Board Room.*

With the significant increase in the power of computers and the ability to store large amounts of data, it is not difficult to imagine how identical

*From an article written by an independent consulting firm, Savage Associates, Wellesley MA. "CIM The Organizational Issues, Fifth Generation Technology and Second Generation Organizations."

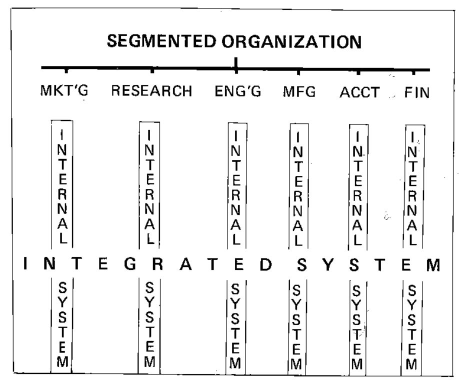


FIGURE 1 Segmented organization.

data can be used more effectively by persons in a host of different departments. This can be illustrated by placing persons from such departments around a table as shown in Figure 2. This critical mass of people do the following:

1. Determine whether a design requirement for a new order is available through a retrieval system.

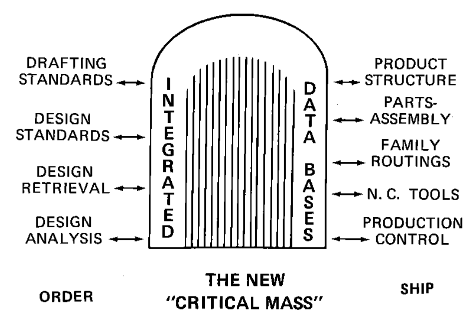


FIGURE 2 The critical mass.

2. If none is available, use design analysis to create a new design to meet customer requirements.
3. Apply design standards based on company requirements.
4. Apply correct drafting standards to the new part and assembly designs.
5. Prepare a bill of materials for the new product.
6. Prepare the geometry for the parts and assembly of the product.
7. Utilize a family routing for similar parts to prepare a specific routing for the new designs.
8. Prepare the NC tool path for the new geometry.
9. Enter the data for the parts and assembly into the MRP and the production scheduling system.

This "round table" of people represents a "critical mass" of activities performed today in separate rooms with separate data, sequentially, over many weeks.

By adding one ingredient, a data base, all of the persons involved may now perform their tasks in parallel rather than sequentially and with the same, not repeated, data. In this atmosphere, organizational barriers cannot exist as they once did.

Following are a few paragraphs from a book written by Dr. Joseph Harrington, Jr. (a consultant with Arthur D. Little), *Computer Integrated Manufacturing*, Robert E. Krieger Publishing Company, Huntington, New York, 1979.

> There are hundreds, if not thousands, of compartments in the organizational structure of a typical large manufacturing company. While it is customary to find the lines of authority in an organization following a pyramidal structure (frequently known as the military structure), it is not uncommon to find that many managers have technical authority over the activities of groups reporting to another manager. This fragmentation of management is far from desirable even though it seems inevitable if very large organizations are to have an effective form of management.
>
> For a century and a half American industry has been teaching its middle managers that it is their responsibility to maximize the productivity, efficiency, and output of their individual sectors.... But optimizing the subdivision does not guarantee that the total organization will operate optimally. On the contrary, while management structures can be fragmented as much as one wishes, the manufacturing function carried out under that management still remains a monolithic unit. If such a sturcture is to be optimized, it must be a totality and not piece by piece. Furthermore, these independent domains are inherent barriers to communication. While information theoretically should flow freely from point of origin to point of use anywhere in an organization, all too frequently it is forced to follow the line structure of the organization up through the hierarchy to a common point and then down again to the destination.
>
> *There are many reasons for such restrictions, the principal ones being the human traits of self-protection and desire to control all the elements of one's own area.*

> Business school professors have been cautioning against this system for years. It's defects are obvious to the impartial observer and libraries are full of documents pointing out the hazards encountered. In a paper by Wickham Skinner, Professor of the Harvard Graduate School of Business Administration, to the Technical Council of the Numerical Control Society, in April 1970, he stated, "The American factory system is obsolete."
>
> The last two decades have seen the introduction of new technologies that not only revolutionize the manufacturing process but virtually demand the revolution of the management process that go with them.... As can be seen, the web of information exchange reaches into every part of the manufacturing organization and electronically circumvents the old communication problems and all their inherent barriers... Obviously, the new system will not be dominated by a single person, such as the craftsmen of colonial days, but the same responsibilities will be fulfilled by an *integrated group of persons*.

How then shall *we* meet the challenges that new technology presents to our organizations? Can we face them objectively, critically, and creatively? Our skill in solving this problem will be a major factor in the success of our future manufacturing efficiency and, perhaps, in our very survival.

Emotions

We have discussed at some length the implied necessity for a new organizational structure. The impact on all persons involved can be highly emotional. Many managers will feel that they have lost their "territory" and even their status. It may at times be difficult for personnel to know the exact function under which they operate and report.

Certainly, today, one of the most significant problems of implementing a simple CAD/CAM system is organization. To whom should computer operations report—to Engineering—to Data Processing? To whom should the designers and/or drafters on terminals report—to the Director of CIM—to Engineering? To whom should the parts programmers report—to the Director of CIM—to Manufacturing—to Engineering? Many other unresolved problems of organization have and will create some fear, anxiety, anger, and frustration.

Knowledge

There seems to be no question that technical knowledge is outpacing our ability to utilize it. Even as this article is being read, both the knowledge of the reader and that of the writer concerning CIM is becoming obsolete. A quote from an article in the March 1983 *Atlantic Monthly*, "The Next American Frontier," by Robert B. Reich, Kennedy School of Government at Harvard, clarifies this point:

> A flexible manufacturing system requires an organization designed for change and adaptability; high volume, standardized production requires an organization geared to stability.
>
> A flexible manufacturing system is rooted in discovering and solving new problems; high volume standardized production basically involves routinizing the solutions to old problems.

The abundance of new knowledge will continually alter our concepts of manufacturing. New knowledge will change our design methods, parameters, and product specifications. For example, NASA has proposed that steel with special new characteristics be rolled in space. How will steel mill manufacturers face this challenge?

How will we educate our drafters and our engineers? How will top and middle management become accustomed to managing these new and powerful techniques?

Data and Flow

For the past 20 years, data have been "flowing" more or less efficiently and effectively through a series of data processing business systems. Unfortunately, as mentioned previously, engineering documentation has often been provided by other than the Engineering Department. With the implementation of a significant new addition to this information and flow, part and assembly geometry, a new version and philosophy of data flow must be pictured. No longer can the Engineering Department sit by idly as others transform their data into computer information. The Engineering Department is now part of the day-to-day flow of information. In fact, it is the originator of much of it.

Also, as illustrated graphically, use of a data base will create an atmosphere where on-line systems used interactively will be mandatory. It should be remembered that the implementation of a CAD/CAM system and CIM, including the graphics and design analysis portion, will be successful only when all users understand and willingly embrace the new technologies.

How will we create enthusiasm in an Engineering Department to obtain its cooperation? How will other departments accept data from Engineering without reinventing it? Will Manufacturing Engineering be satisfied with the accuracy of geometry and other data created in Engineering?

OLD GEEZERS—YOUNG TIGERS

Young "tigers" and old "geezers" describe different types of people to each of us. We should use these terms not to describe a chronological age but a way of thinking—a philosophy used in addressing the day-to-day workplace. The implementation of new technologies has created a new problem.

It is possible for a company to believe that a young enthusiast, a young tiger with much knowledge about the new technologies, be placed in a position of responsibility and authority to implement them. Unfortunately, this person may make little use of past knowledge, problems, company history, and other significant background material.

On the other hand, to make certain that this past background is not destroyed, an experienced person within the company may be assigned the responsibility and authority. Unfortunately, this person may be uncomfortable with the new technology—perhaps even antagonistic to its use.

In either case, the company may not be well served. How can we merge the best of both? How can we take a mixed group of old tigers and young geezers so that they will integrate the best of the old and the new? This may well be the most important consideration and challenge of all.

A NEW LOOK AT OLD TECHNOLOGIES

For the past several decades, members of management have tended to be uninvolved in the solution of "minor" systems problems. This lack of participation has created incomplete or parochial solutions. Now this lack of participation is showing up in the need to modernize present systems before investing time and money in more sophisticated and exacting new ones.

It is inevitable that members of management must now participate in solutions, because most of the problems they represent are not departmental but corporate in scope. These details are not glamorous. Often they cannot be evaluated clearly in economic terms. Nevertheless, many of them have delayed modernization for a long time and must be addressed or an integrated CIM system may be impossible to implement. Let us discuss a few of them:

Part numbers.
Piece part drawings.
Bill of materials.
Modular design.
Design retrieval.
Part families.

Part Numbers

Almost 50% of companies in the United States have job-oriented and not part-number-oriented systems, and the part-number systems in a proportional number of companies are out of date with obsolete built-in significance and too long and/or a difficult mixture of alphanumeric symbols.

Job-number systems were impressed upon Manufacturing to make certain that the cost of an order was collected. Because job number accuracy was accentuated, job numbers soon replaced part numbers in significance within the system. Thus, it is possible to find identical designs with identical part numbers being manufactured separately under different job numbers. Two cases may occur: Job numbers within the same customer order will be different, hiding duplicate parts; and job numbers between orders will be different, hiding duplicate parts. Integration of requirements, manufacturing planning, family routings, and design retrieval cannot increase productivity if this artificial segmentation is continued. Thus, the Accounting Department, as well as Engineering and Manufacturing, must convert to a system that makes the part number the means to retrieve data and combine requirements.

Many part numbers are too long because they been stuffed with significant data that is useful to only a few, is often redundant, and may be obtained and maintained more easily in other ways. (The writer has seen a part-number system requiring 45 digits!) Streamlining activities from the creation of a new design in a CAD/CAM system to its manufacture on an NC machine tool requires a system that can be implemented rapidly and used as a tool and not a millstone. In addition, far greater access to engineering documentation, from design geometry to engineering drawings and parts lists, requires a simple means of obtaining the data. Recognizing that a part number is the "key" to all other data concerning the design, that key—the part number—should be short and easy to use.

It will not be easy to change a part-number system to make it reliable and more important than the job number, but productivity demands it. Few conversions have been successful without management participation, including awareness, understanding, and agreement.

Piece Part Drawings

A large segment of today's manufacturing companies also have multipart drawings and assemblies. These companies should convert to piece part drawings. Eleven reasons for such a conversion exist:

1. Clarity of fabrication details
2. Economical part design
3. Documentation changes
4. Part reusability
5. Design retrieval systems
6. Elimination of duplication
7. Raw material requirements
8. Principles of manufacturing
   A. Routing requirements
   B. Tooling requirements
   C. Group technology
9. Family drawing preparation
10. Computer-aided drafting
11. Purchased part documentation

Personnel within the Engineering Department of a company who do not now have piece part drawings tend to resist this change, marshaling emotional forces rather than logic. Such statements as "It will create more paper," "It will take more time," and "It will cost more" are heard. The corresponding benefits as outlined in the 11 reasons for the conversion are rarely heard. Members of management must participate in the resolution of this problem to avoid endless discussions and long delays.

Bill of Materials

Job-number systems initiated by accounting departments were, at one time, a worthy alternate, or replacement, for a bill of material system. It must be remembered that the dsicipline of job-number accounting met the requirements of an accurate costing system. However, a mechanized bill of materials performs all the functions for which job costing was designed and meets many additional integrated system requirements that a job-number system cannot hope to achieve.

A bill of materials is an accurate and rapid means to record part and assembly relationships in an engineering design. In addition, it will eventually be the data base method to store and retrieve geometric data for part and assembly designs. It has already become a mandatory system in the implementation of MRP and other mechanized engineering documentation systems. Thus, it is imperative that a bill of materials be installed and/or enhanced within a manufacturing company.

Modular Design

A significant advance in product documentation occurs when modular design techniques are used. The principle of modular design is to extract all design variables from the product and arrange them into separate bills of materials known as "attachments." These variable parts may also be moved to higher levels in the old bill of materials or split into segmented assemblies. The remainder or "constant" poriton of the product is then divided into major assemblies, often called "modules." It should be emphasized that the physical design must be converted, if necessary, to match this division. Several important results are obtained with this procedure:

1. Modules and attachments are used on a master schedule to generate MRP.
2. Modules and attachments are combined based on a customer order, thus eliminating a significant number of bills of materials for slight variations to a product. It might be best explained by stating that a product is documented by part numbers that are expanded in quantity, arithmetically utilizing modular design rather than increasing geometrically as required when each variation is documented with a specific bill of materials.
3. Less inventory is required to maintain many product variations because the modules and attachments may be stocked in preparation for an order rather than for all possible combinations.
4. More standardization of products is possible because each module is designed to mate with other modules and attachments. Minor variations do not require complete new documentation—only that which shows changes.
5. Production can be increased in that the modules can now be identified and produced together, where previously identical assemblies and parts may have been hidden under different bill of materials part numbers and under several variations within them.

Design Retrieval

The most significant tool for successful implementation and integration of CIM and CAD/CAM is a classification and design retrieval system. Often thought of as only an engineering tool, management must understand the principles of these systems in order to see their corporate implications and importance.

Classification systems within a manufacturing system can be divided into four segments: raw material, purchase parts, designed parts, and assemblies and products. Let us look for a moment at the designed part segment.

Using a classification system, designed parts are divided into "families" with similar geometric shape, form, or functional relationships. One simple family, for example, may be those designs that are metallic, round, multiple diameter, with diameters successively decreasing, or with a through-going center hole. This family will contain 15 to 20 parts if the system is properly designed. The importance of part families for productivity improvement and production standardization are discussed below.

A design retrieval system will have many uses and economic benefits. Some of them are as follows:

1. Design retrieval statistics have shown that from 4% to 20% of new designs already exist if they can be retrieved. Because the cost of creating a new design may vary from $12,000 for a complex forging to $200 for a simple fabricated part, elimination of design redundancy generates a significant savings. (Many companies who have implemented a design retrieval system have paid for it in less than a year.)
2. Uniformity of data is encouraged by use of a design retrieval system. This also allows a more detailed description of purchased parts, avoiding one-vendor restrictions.
3. When designs are classified using permanent form, shape, or functional features, an important part family data base can be established. These data are the key to the inetgration of CIM.

Engineers (and others) tend to resist the implementation of a design retrieval system. Thus, management must assist in developing a climate conducive for such a system.

Part Families

A direct and important result of a classification system is part family data. Suppose one was asked to look at 80,000 designs and establish drafting standards, routings, and tooling for them. It would be easier, more practical, and more logical to consider them not as individual designs analyzed randomly but rather as 1,000 part families where similar parts with similar features are gathered. It is more prudent, quicker, and more economical to tackle standardization using 1,000 families rather than 80,000 random parts. A properly designed retrieval system will allow this to take place.

Some of the family data that can be generated are as follows:

Design standards.
Drafting standards.
Producibility tips.
Design analysis parameters.
Family processes.
Parts catalogs.
Tooling catalog.
NC cutter tool path.
Parametric drawings and geometry.

These data might be used in the following beneficial ways:

1. A parametric drawing can be used in conjunction with the classification system and a parts catalog. When a new design is required, a look at the appropriate part family and catalog section will make certain no existing design meets the new criteria. Once confirmed, the parametric family drawing is invoked to enter the dimensions and other criteria for the new design, and the computer prepares it automatically in the form of a drawing. (A parametric drawing should be considered a special program that literally draws the geometry of a part. However, the program has voids in it. With appropriate prompts, the user fills the voids by entering all the dimensional data necessary to create a specific design.)
2. The family of parts may be examined to create a family process and routing that optimizes the work centers, setup time and requirements, and other data for the family. This first requires a study of the present routings and processes which, when complete, will result in a more uniform and optimum set of routings. It will also allow all new designs to be released quickly by preparation of a new routing and process through minor alterations to the existing family process and routing.
3. Once a family of parts is established, an index of perishable tools, jigs, and fixtures can be prepared for each of them. This index may be used so that new designs will not generate new tooling until the family of tools index has been examined.
4. The classical use of part families is that of "group technology" or "cellular manufacturing." Using family data as design sizes, weights raw materials, and forecasted usage, machine cells can be developed
5. A family tool path for those operations requiring NC tools will also expedite the preparation of a specific tool path for a new design. In addition, tool packages for an NC tool can be examined to make certain that no additional features, such as nonstandard hole sizes, taps, and so forth, have extended the tool package requirements to create a tool change. This type of standardization is also accentuated in the use of producibility tips.
6. Producibility tips imply a joint agreement between Engineering and Manufacturing in the best means to standardize methods and make manufacturing more economical, higher in quality, and uniform. Such items as standard drills and taps, bevels, bend radii, finishes, and plating, standardized by family, create a fine benchmark for both Engineering and Manufacturing to avoid unnecessary costs and engineering changes. In addition, the producibility tips are not restrictive. They are usually tabulated on the basis of cost, with comparative cost increases also shown should they be changed. This allows designers alternate methods of design but indicates "penalties" or costs of alternate methods.

If we review the details that require management attention, it is apparent that the majority of them are instituted in Engineering. This recognition again emphasizes the need to bring the Engineering activities into the mainstream of data flow in business and CAD/CAM systems. If CIM is to be realized both as a technique and as a truly integrated approach to manufacturing, however, all departments must communicate more readily, faster, more precisely, more honestly, and with management knowledge and approval. The challenge to management is there for all to see.

THE FUTURE

It is important to understand that "future shock" will remain with us. We must learn the lessons of flexibility, continuing education, new management techniques and, above all, we must expand and maintain our sense of humor; the latter is truly important. As soon as we are confident that we

understand some new technology, our knowledge becomes obsolete and, perhaps, worthless. The only certain knowledge that we have today is that constant change is constant. Let us develop our management techniques and actively participate to take advantage of these challenging new ideas and inventions.

GLOSSARY OF TERMS

MIS (management information system). A discipline designed to prepare and present timely data to operating personnel in their performance of daily activities.

CAE (computer-aided engineering). A process that uses a set of algorithms and formulas to design products, parts, and assemblies more rationally, effectively, and economically.

CAD (computer-aided design). A discipline used to capture the geometry of design in a manner in which both Engineering and Manufacturing can use it to create physical parts and assemblies accurately and rapidly.

CAM (computer-aided manufacturing). A process used to convert design entities into physical parts and assemblies using the latest manufacturing techniques as generative process planning, NC tools, and flexible manufacturing or group technology.

MRP (material requirements planning). A technique used to schedule material into the manufacturing process at the proper time and in the proper quantities.

MRPII (manufacturing resource planning). An extension of MRP that adds to the technique of material logistics by scheduling production machines and manpower requirements as an integrated system.

CIM (computer-integrated manufacturing). The integration of all the support systems, with a significant emphasis on an integrated data base to avoid redundancy, increase productivity, and create a synergistic pool of information.

CHARLES S. KNOX

C. ITOH ELECTRONICS, INC.

Established in December, 1973, C. Itoh Electronics, Inc. (CIE), was the successor to a small electronics activity within the parent company, C. Itoh and Comapany (America), Inc.

Beginning in November 1968, marketing of small Japanese-produced printer mechanisms was begun, with the first sale made in May 1969 to Wang Laboratories. These printer mechanisms became some of the first Japanese-manufactured printer devices sold outside of Japan. In the mid-1970s CIE determined that microcomputers in the early stages of development would eventually need a printer that would be price and configuration compatible with low-cost computers. Nick Kondur, an independent consultant was hired to design the first low-cost, desktop 80-column printer in June 1976. When the product was initially put into production, Anadex, a company headquartered in the Los Angeles area, was selected as the manufacturer.

With its partner, Thomas L. Siwecki, and the research facilities at Stanford Research Institute, CIE developed the first optical storage and retrieval system beginning in April 1976. The same team jointly developed one of the earliest printers in 1979.

As the microcomputer industry gained momentum in 1979–1980, CIE provided a low-cost daisy-wheel printer for less than $1000. This allowed the microcomputer manufacturers, for example Exidy and Ohio Scientific, to sell their systems for business applications and thereby compete favorably with dedicated word processors and other dedicated business systems.

In conjunction with Apple, beginning in 1982, CIE helped to develop the first printer for the Macintosh and thereby made desktop publishing available to everyone.

After a 3-year development cycle, CIE delivered the first low-cost ion deposition printer to users who could provide very low-cost, in-house printing. The ion printer provides users with the capability of obviating the need to use offset or similar kinds of printing equipment.

CIE is a company with sales of more than $500 million. It is a significant exporter of U.S.-manufactured, high-technology products to Japan, in addition to developing and importing offshore-manufactured products. Through its affiliated companies, it is in the satellite communication, fiber optics, transoceanic communication, and value-added network industries.

ROBERT J. COWAN

CCITT

CCITT, the International Telegraph and Telephone Consultative Committee, is a committee of the International Telecommunications Union (ITU), which is a specialized agency of the United Nations Organization. The ITU, which is located in Geneva, Switzerland, is composed of five subunits: the General Secretariat, the Administrative Council, the International Frequency Registration Board (IFRB), the CCITT, and the CCIR, the International Radio Consultative Committee.

The CCITT's purpose is to recommend standards for communication systems. The goal of these standards is to provide a specification for interconnecting existing telecommunication networks throughout the world and to define a uniform access mechanism to these networks. To date, approximately 2,000 recommendations for standards covering all aspects of telecommunications have been made by CCITT.

About 160 nations participate in the CCITT. They are generally represented by delegations formed by their governmental agencies responsible for telecommunications. For most countries these agencies are their postal, telegraph, and telephone (PTT) administrations. Historically, the United States has participated in CCITT through representatives of AT&T with State Department oversight. In recent years, the Federal Communications Commission (FCC) and the National Telecommunications and Information Administration have become increasingly involved. These official or administration members of CCITT vote on the proposed recommendations.

Telecommunications users and suppliers of communication equipment and services are thus not directly involved in the approval of recommendations. However, they can participate in the process of developing a recommendation through consultation with their national delegation. In addition, four other classes of membership to CCITT are available for nongovernmental participators or observers. Recognized private operating agencies (RPOA) are public or private companies that supply telecommunications services (i.e., AT&T, ITT). CCITT also recognizes some users, manufacturers, and other organizations as scientific and industrial organizations (SIOs). Liaisons are maintained between CCITT and other groups, such as the International Organization for Standardization (ISO), the European Computer Manufacturers Association (ECMA), and the International Telecommunications User's Group (INTUG). Finally, special treaty agencies such as the World Meteorological Organization, which have a vested interest in telecommunications, can participate in CCITT. These groups do not participate directly in the voting on recommendations but contribute greatly to their formulation.

The supreme body of the CCITT is the Plenary Assembly, which is made up of all interested administration members and any RPOA with the approval of an administration member. The Plenary Assembly meets approximately every 4 years and identifies areas of interest and formulates questions that it wishes to study. Study groups are created by the assembly, and questions are assigned to each group for investigation (Table 1).

TABLE 1 CCITT Study Groups

I	Telegraph operation and tariff
II	Telephone operation and tariff
III	General tariff principles
IV	Transmission maintenance of international lines, circuits, and chains
V	Protection against electromagnetic disturbances
VI	Protection and specifications of cable sheaths and poles
VII	New networks for data transmission
VIII	Telegraph and data terminal equipment; local connecting lines
IX	Telegraph transmission quality; specification of equipment and rules for the maintenance of telegraph channels
X	Telegraph switching
XI	Telephone switching and signaling
XII	Telephone transmission performance and local telephone networks
XIII	Automatic and semiautomatic telephone networks
XIV	Facsimile telegraph transmission and equipment
XV	Transmission systems
XVI	Telephone circuits
XVII	Data transmission
XVIII	Digital networks

Much of the work of CCITT is done in study groups. Often working parties are formed from the study group to focus on specific sets of questions. Special rapporteurs may be assigned to coordinate the investigation of specific points of study. The study groups hold international and regional meetings over a study period of 4 years to discuss and attempt to resolve the questions that they have been given. The current study period began in 1985.

At the end of the study period, the recommendations of the study groups are voted on at a Plenary Assembly of the CCITT and, if accepted, are published as official CCITT standard recommendations (Table 2). The multivolume book of recommendations for each study period is given a color designation (i.e., "the yellow book"). Recommendations may be altered or updated by a study group in a subsequent study period.

The official languages of the ITU are French, English, Spanish, Chinese, and Russian, the first three of which are designated as working languages. All CCITT publications are printed in French, with most of them also published in English and Spanish editions.

CCITT recommendations are generally implemented as standards in Europe. The United States and several other countries consider the standards recommendations to have the status of treaties that must be ratified by their respective legislative bodies before adoption. In the United States, existing standards from groups such as the Electronic Industries Association (EIA),

TABLE 2 CCITT Recommendations Series

A	Organization of the work in CCITT
B	Means of expression
D	Lease of circuits
E	Telephone operations and tariffs
F	Telegraph operation and tariffs
G	Telephone transmission on metallic lines, radio links, satellite, and radiotelephone systems
H	Lines used for the transmission of signals other than telephone signals, such as telegraph, facsimile, data, etc.
I	Integrated services digital networks
J	Program and television transmission
K	Protection against interference
L	Protection of cable sheaths and poles
M	Maintenance: telephony, telegraphy, and data transmission
N	Maintenance: sound program and television transmission
P	Quality of telephone transmission
Q	Signaling and switching
R	Telegraph
S	Printing telegraph equipment
T	Facsimile
U	Telegraph switching
V	Data transmission
X	Public data networks
Z	Software

The Institute of Electrical and Electronic Engineers (IEEE), and the American National Standards Institute (ANSI) have remained dominant. In newer areas of communication applications, the CCITT recommendations have seen an increasing rate of adoption by American manufacturers of communications equipment and suppliers of communications services.

Of primary interest are the CCITT recommendations for data communications that have been developed by two CCITT study groups (Table 3). Study group VII is responsible for recommending standards for data communication over public data networks. Recommendations developed by this group are given identifiers with the prefix "X" (i.e., X.21, X.25, etc.). These are termed the "X" series recommendations. Study group XVII has the task of recommending standards for data communication over telephone systems. These recommendations have a "V" prefix and are called the "V" series recommendations (i.e., V.21 and V.24).

TABLE 3 Sample CCITT Recommendations

V.21	300 bits per second duplex modem standardized for use in the general switched telephone network.
V.24	List of definitions for interchange circuits between data terminal equipment (DTE) and data circuit-terminating equipment (DCE).
V.54	Loop test devices for modems.
X.21	Interface between DTE and DCE for synchronous operation on public data networks.
X.25	Interface between DTE and DCE for terminals operating in the packet mode on public data networks.
X.75	Terminal and transit call control procedures and data transfer system on international circuits between packet-switched data networks.

The current CCITT work that seems to have attracted the most attention from the telecommunications community is the Integrated Services Digital Network (ISDN) standards development. ISDN's will combine voice, data, and video traffic over a single digital network. These networks will be composed of wire, radio, satellite, and optical links. Other ongoing CCITT work focuses on the development of standards for languages for the description, design, and control of telecommunication networks.

Note: CCITT publications may be obtained by contacting

General Secretariat
International Telecommunications Union
Place des Nations
1211 Geneva 20
Switzerland

BIBLIOGRAPHY

Bertine, H. V., "Physical Level Interfaces and Protocols," in *Data Communications Network Interfacing and Protocols*, IEEE, 1981, pp.2-1–2-18.

Dickson, G. J., and DeChazal, P. E., "Status of CCITT Description Techniques and Application to Protocol Specification," *Proc. IEEE*, 71,(12) (December 1983).

Freeman, Roger L., *Telecommunication Transmission Handbook*, Wiley, New York, 1975.

Lewin, L., ed., *Telecommunications: An Interdisciplinary Survey*, Artech House, Dedham, MA, 1980.

Levillion, M. E., "Involvement of Users and Manufacturers in the Development of CCITT Packet Switching Recommendations," *IEEE Trans. Commun.*, (February 1984).

Linington, P. F., "Progress in Open Systems Standardization," *Interfaces Comput.*, 2, 205–220 (1984).

U.S. Senate Commitee on Commerce, Science, and Transportation, *Long-range Goals in International Telecommunications and Information: An outline for United States Policy*, U.S. Government Printing Office, Washington, D.C., 1983.
Yearbook of the United Nations, The United Nations, New York, 1983.

KENNETH SOCHATS

CENTURY ANALYSIS, INC.

Century Analysis, Inc. (CAI), is a California corporation having its headquarters 30 miles east of San Francisco in Pacheco, California.

Founded in 1975, CAI today is a recognized leader in the worldwide information-processing community as a provider of high-quality systems software and productivity tools for large-scale, mainframe-oriented communications environments. The company has been awarded numerous product performance awards for its communications and software offerings, as well as top industry ratings for user-product acceptance.

Although it may seem that a background in mainframe computing would have little relevance in microcomputing, it is actually the contrary. Although CAI started purely as a mainframe software developer, it soon came to realize that "information management" necessitated more than one level of processing; what it required was four levels: personal, departmental, networked, and mainframe.

Beginning with the need to proliferate the mainframe's capabilities to end users within an organization, CAI became heavily involved in the combination of 16-bit microprocessor technologies and office automation systems distributed in a large-scale telecommunications network. The vehicle chosen to front end to the mainframe was a microcomputer, used as an intelligent communications processor to cost effectively service these large terminal networks. This product, called OSI (Office Systems Interface), later became the heart of still another CAI offering, DeskNet, a hardware/software product that provides the ability to run mainframe software and CAI-developed word processing and spreadsheet applications in a multiprocessing, multiwindow environment, providing the end user with a friendly and unintimidating view of information handling through one consistent workstation.

As a result of these ventures, CAI entered a new realm-culminating its several hundred man-years of combined experience in information processing software and mainframe and microtechnology into a general business office automation product called OFFICEWARE.

OFFICEWARE is an integrated office automation system designed to run on IBM's PC product line. It was designed to meet the present and future needs of the office environment by providing comprehensive, integrated, and easy-to-use information-processing capabilities that span personal, departmental, network, and mainframe processing.

In a single-user, hard-disk-based PC environment (see Fig. 1), OFFICEWARE operates under PC-DOS and provides the user with a multiprocessing window-based operating environment and a full set of integrated personal tools, including word processing, spreadsheet, business graphics, list management, calendering, things to do, reminders, and an electronic filing system. It also includes a built-in terminal emulator to simultaneously access non-OFFICEWARE information-processing resources.

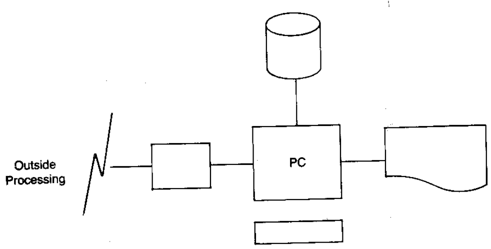


FIGURE 1. *Single-user OFFICEWARE.*

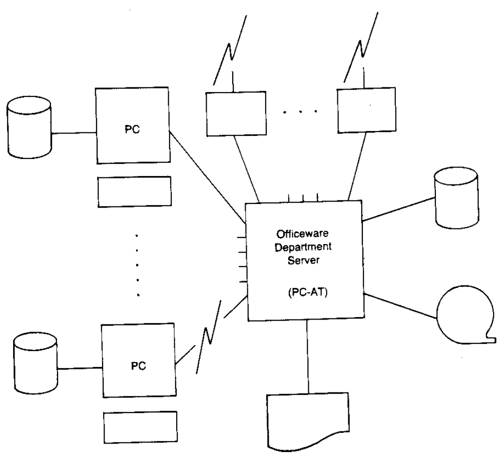


FIGURE 2. *OFFICEWARE basic network.*

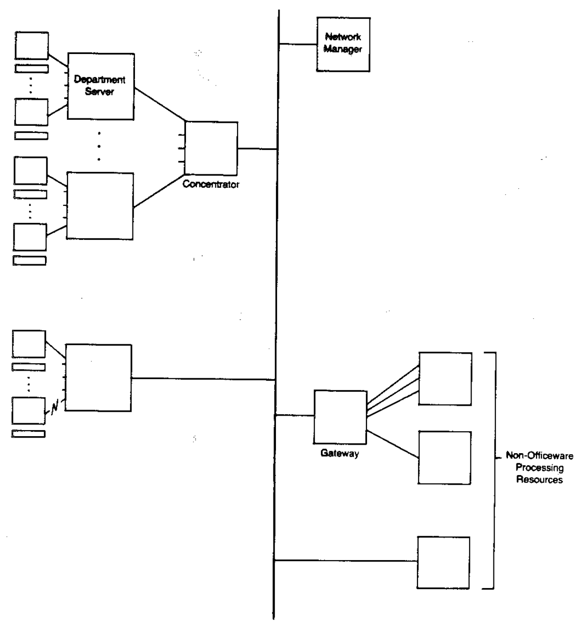


FIGURE 3. *OFFICEWARE expanded network.*

In a basic network (see Fig. 2), PCs are connected using a star topology to a PC/AT running XENIX. Individual users are provided with an additional set of interuser network facilities. Phone and free-form messages are easily forwarded. A "send" function allows documents to be transferred in final and/or in revisable form. The personal calendar becomes a networked calendar that automatically checks other users' schedules and interfaces to the mail application to deliver scheduling information. A central print server allows printer sharing to take place. A document-archiving subsystem allows sharing access to multiuser departmental applications running on the PC/AT. A network monitoring subsystem immediately informs users of network traffic requiring their attention no matter what work is currently active.

In an expanded network (see Fig. 3), an Ethernet local area network (LAN) is introduced to provide users access to other processors (e.g., existing mainframe) using a gateway. PCs can be either directly connected and/or CAI concentrator-attached to the LAN. Non-OFFICEWARE computers can be either interface capabilities. The end user is provided with a new window type that allows simultaneous desktop access to these added resources.

The desktop is the key to OFFICEWARE's power, with such features as windows, pull-down selection lists, soft keys, across-applications cut/paste, and context-sensitive help. OFFICEWARE's multiprocessing operating environment and applications were all written by the same development team using the OFFICEWARE Toolkit; the result is such unparalleled ease of learning and ease of use that it is not uncommon for a new user to become fully productive without looking at the documentation.

The expanding PC product line, with its memory, compute power, disk storage, and ability to be networked, proved to be an excellent environment for the product. This multilayered open architecture provided a flexible, comprehensive, cost effective solution for information processing and office automation while being adaptable to new developments as they take place in the inforamtion-processing industry.

CAI's investment in OFFICEWARE development currently stands at well over $7 million. Significant resources were committed to ensuring that the different types of processing (personal, departmental, network, and mainframe), which CAI recognizes must be available as office automation, become a requirement, rather than an option, in the modern business environment.

SARA LAFRANCE

Editor's Note: Since this was written, OFFICEWARE has evolved from simply integrated office automation into a network management system. The microprocessor still functions as an intelligent of "smart" workstation, but rather than standalone or attached to a LAN, now participates in an extensive, multi-computer network. For more information, the reader should contact: Century Analysis Inc., 114 Center Ave., Pacheco, California 94553.

CHEMICAL ENGINEERING, MICROCOMPUTERS IN

INTRODUCTION

This article, reviewing the role of microcomputers in chemical engineering, can be compared to a snapshot taken of a rapidly moving object; the details are blurred, but the general outline is clear. The moving object, of course, is digital technology. Forty years ago, when the first electronic digital computers were developed, it would have been impossible to have predicted how rapidly those early behemoths with their thousands of vacuum tubes, huge power supplies, and air conditioning systems would be replaced by solid-state technology with orders of magnitude reduction in size, operating cost, and price. It is equally dangerous today to predict just where future developments will occur. However, it is important to realize that from the standpoint of the computer user, it is the tools that are changing and not the fundamental principles of the discipline. This article will, therefore, emphasize the ways in which microcomputers are being used and can be used in chemical engineering practice and is not intended to serve as a buying guide for hardware or software. It is written to inform the practicing chemical engineer about microcomputer applications and to provide an overview of computer applications in chemical engineering for the computer specialist.

The term microcomputer is defined in different ways. As used here, it refers to any computer that is contained on a single board or the equivalent of a single board. It includes personal or desktop computers, as well as special-purpose instrumentation and control computers. Because of advances in computer hardware, the computing ability of the microcomputer is being expanded continually. The discussion that follows will therefore include reference to applications that are not now practical with microcomputers but that may become so in the near future.

MICROCOMPUTER AIDS FOR DESIGN AND SIMULATION

One of the most important functions of the chemical engineer is the design of new chemical plants and the modification of the operation of existing plants. Process design generally refers to the development of a new process, and process simulation, to the analysis of an existing process. Because the former requires the consideration of alternative design structures and the selection of a best design, it is open ended. The latter requires that a mathematical model for the existing process be developed and then used to study and improve process performance. A typical plant consists of a large number of processing units through each of which material is continously flowing. Processing units are usually interdependent, and a change in one unit affects many other units. Some operations involve toxic materials and are carried out at high temperatures and high pressures, and safety must be a continual concern.

An important part of the process engineer's work is the calculation of unit performance. This may involve the design of a new unit, a modification of an existing unit, or the effect of a change in some operating condition (perhaps a flow rate, a temperature, or a pressure) on the operation of the unit. At a higher level, the integration of individual process units into a complete process must be carried out with increased consideration for economic and social factors. The way in which all of these calculations are being carried out is undergoing revolutionary changes.

The Desktop Personal Computer

Thirty years ago, the working tool of the process engineer was the slide rule—an analog device for multiplying and dividing with accuracy to about three significant figures. Mechanical calculators or large mainframe computers could be used for problems requiring more precision. About 20 years ago, the first microcomputers in the form of the hand-held calculator came into widespread use. These early calculators with 8 to 12 significant figures, built-in trigonometric functions, and one or two memory cells were in themselves a great advance in computational ability, but they are primitive tools compared with the present day personal computer. At a modest cost, the process engineer now has on his desktop a computer that is more powerful than mainframe computers of 20 years ago. It will have about 500Kb of random access memory, perhaps 10Mb of disk storage, communication facilities for access to other computers, and computer programs that can be used to solve many of the routine process design problems that arise. It has been estimated that three out of four engineers will be using a personal computer on the job by 1990 [*1*].

Design of Process Modules

A complete chemical process can be broken down into individual steps, called process modules or unit operations. The design of an individual module is a standard procedure. The calculations for a typical process module require the solution of a set of simultaneous equations describing the performance of the unit. The equations describe the physical phenomena that govern the operation of the unit, as well as material balances and energy balances made about the unit. The complete set of equations constitutes a mathematical model for the unit operation and may include economic factors such as construction and operating costs. If the values of the process variables (e.g., temperatures, pressures, and flows) do not change in time, the process is said to be at steady state, and the equations will be a set of simultaneous algebraic nonlinear equations. If the values of the process variables change in time, then the process is said to be dynamic, and the equations will be a set of simultaneous nonlinear differential equations. Most process units operate at steady state, and the steady-state equations are usually used as the basis for the design. The discussion in this section is limited to the solution of steady-state equations. However, the dynamic characteristics of a process are very important and must be considered in the analysis of control systems, plant start-up and shutdown, and response to emergencies. The solution of dynamic equations will be treated later when control systems are discussed.

A rigorous description of a process module may easily be a hundred or more equations, and if the equations are nonlinear, as is usually the case, an

iterative solution method is needed. Less rigorous descriptions are, of course, available and frequently used. The availability of computers has led to increased use of more complicated mathematical models for process units, even in some cases when simpler models would do as well. Just as work expands to fill the time available, computational requirements will expand to saturate the available computing facilities.

Some typical process modules used in the chemical industry are (*a*) distillation columns, (*b*) absorption and stripping columns, (*c*) chemical reactors, (*d*) heat exchangers, (*e*) solid-liquid separators, and (*f*) piping networks.

Microcomputer software is available for these processes and for many others. Listings of available programs can be found in publications of the American Institute of Chemical Engineers [*2*], *Chemical Engineering* (McGraw-Hill, New York), *The Software Catalog* [*3*], publications of CACHE Corporation (Austin, TX), and others (*4–6*). In addition, many corporations have developed their own programs, which are not available to the public. New programs are being added continually. As with all software, the user must make sure that the program will run on his/her computer, will solve his/her problem, and that the seller will provide technical assistance if needed.

Process Flowsheeting

As indicated above, a complete process is composed of many interconnected process modules, and the design of a complete process must take this into account. There is often a strong interaction or coupling between modules because of recycle, which is the recirculation and subsequent reprocessing of unreacted material or undesired products. The need for recycle arises from thermodynamic and kinetic limitations and not from process inefficiency. Another kind of interaction arises because of interchanges of energy between units—for example, the use of a heat exchanger to cool a hot product stream with a cold feed stream. Because of these interactions, the performance of one unit cannot be considered in isolation from other units.

The equations for a complete process can be obtained by taking the equations for the individual process units and linking them together properly. This is sometimes called flowsheeting. The solution of the resultant sets of simultaneous equations can be difficult in spite of the fact that algorithms for solution of the individual units may be well developed. One method is to create subroutines for each kind of process unit and then write an executive program that will call the subroutines as needed. Each subroutine is written with standard inputs and standard outputs. If, when a subroutine is called, all the input information for the unit is not available (usually because the missing input is a recycle stream), it becomes necessary to assume trial values for the missing inputs and to correct them in subsequent iterations. A good executive program minimizes the number of iteration variables by careful analysis of the flowsheet and by making the order of calculation correspond, as much as possible, with the flow of material through the process. This is sometimes called the "sequential-modular" method [*7*]. An alternative and theoretically better, but more complicated, method is to solve the equations for units and interconnections simultaneously—called equation-oriented flowsheeting [*8*].

Flowsheeting programs can be classified by the scope of the problems that can be solved. One very effective approach for preliminary design problems that emphasize mass and energy balances is to use a microcomputer

and one of the many spreadsheet programs that are available [9]. Although spreadsheets are excellent for some purposes, they require some programming and are not applicable to many processes. For more complicated systems, it is desirable to use programs developed for analysis of chemical plants. The simplest kind of flowsheet program is one designed for a particular process or class of processes. An example is the flowsheeting program for coal preparation plants, which was developed at the University of Pittsburgh for the Department of Energy [10]. It can be used to design a new coal preparation plant or to simulate the performance of an existing plant. Modules are provided for each of the common unit operations in a coal preparation plant, such as crushing, screening, washing, and dewatering. The program was developed over a 10-year period and was originally designed to run on a mainframe computer in a batch-processing environment. During this time, however, development in computer hardware were such that the program can now be run on a microcomputer. A typical plant configuration requires about 8 minutes on an IBM PC-AT equipped with a hard disk and a numerical coprocessor.

General-purpose flowsheeting programs are distinguished from special-purpose programs in that they can solve larger problems, have provision for more kinds of unit operations, have more sophisticated methods for convergence of recycle calculations, and have larger data bases for physical properties and other information needed in the design. They generally require that a minicomputer or mainframe computer be used, but this will certainly change in the near future as versions become available for microcomputers. Indictive of this trend is the recent availability of HYSIM, an interactive simulator [11] that can be run on a personal computer. Some widely used general-purpose flowsheeting programs that are commercially available are PROCESS (Simulation, Sciences, Inc., Fullerton, CA) and ASPEN (Aspen Technology, Inc., Cambridge, MA).

Plant Design

Process design, as described above, is only one step in a complete design. It is used to establish a desirable configuration for the plant and to fix important operating parameters. To carry out the process design, a plant design is necessary. It will include detailed process flow diagrams, engineering drawings showing location of all equipment at the plant site, piping and instrumentation diagrams, and equipment descriptions suitable for use in purchase orders. Plant design requires close cooperation between the chemical engineer and other engineering disciplines, such as structural engineering, mechanical engineering, and electrical engineering. There is a heavy emphasis on the preparation of working drawings so any computer program for plant design must include good graphics, preferably with the capability for three-dimensional representation. Commercially available computer-aided design (CAD) packages are commonly used. These systems are generally mainframe or minicomputer based, with microcomputers used as terminals [12]. At present, CAD systems based on stand-alone microcomputers have limited capabilities [13], although it is clear that this will change with future hardware and software developments.

Integrated Engineering Design

The advantages of coordinating process design and plant design are obvious. The specifications obtained in the process design are needed for the plant

design, and much time and effort can be saved if they can be transferred efficiently from one activity to the other. The terms "computer-aided engineering" and "integrated computer-aided engineering" are used to describe systems that can accomplish this. Rather than develop new systems that can do all these functions, the emphasis today is on integration of existing design packages. Two approaches are used [14]. In one, existing packages are modified so that they can interface with the particular data base that contains the desired information [15]. A second approach is to interface all design packages with a general-purpose data base. Although more appealing theoretically, realistic implementation of the latter would require industry standards for representation of data. For example, the detailed process flowsheet is the basic document for any plant. It is the basis for construction of the plant, and it must be changed whenever a process change is made. It would obviously be highly desirable not only to keep this as a printed document but also to maintain it in a form that can be readily accessed by a computer. There are many difficulties in doing this, because the flowsheet is not only a graph but contains much nongraphical information that is vital to the process—information such as equipment specifications, equipment suppliers, and design levels for process variables. Keeping in mind that the process flowsheet will be needed for many years, it is obvious that a standard representation that would be independent of the specific CAD package or computer system is needed. One standard that CAD users in the United States are starting to adopt is the National Bureau of Standards' Initial Graphics Exchange Specification, (IGES) [16]. Different standards are used in other countries. The International Standards Organization (ISO) is now working on an international standard called STEP, Standard for the Exchange of Product Data, which may be widely used in the future [17].

MICROCOMPUTERS FOR CHEMICAL ENGINEERING DATA BASES

In one sense, any collection of information is a data base. However, the term is usually restricted to "a shared, structured collection of data capable of existing between program runs, together with a definition of the structure to which the data base conforms" [18]. Application programs can access a data base to obtain needed information and can store results in the same or in a different data base. One important chemical engineering application for data bases is in the integration of plant and process design as described above. Two other applications of great importance are bibliographic data bases and physical property data bases, which are described below. These data bases are usually large and require more memory than is available in a microcomputer. As a result, microcomputers are used principally as terminals to access mainframe or minicomputers. One technical development that will make these data bases directly accessible to microcomputers is the compact disk (CD), which is already well established for music reproduction. A single disk can hold more than 40 Mb of data and would be a logical peripheral for a microprocessor workstation.

Bibliographic Data Bases

Most bibliographic data bases require a large mainframe computer and are accessed interactively from microcomputer terminals at remote locations. They require constant updating to keep them current and use sophisticated

programs for locating the desired information. The most comprehensive data base for the chemical literature is maintained by the American Chemical Society through the Chemical Abstracts (CA) Service (Columbus, OH). More than 14,000 periodicals, which publish chemically related material, are regularly abstracted, and three on-line data bases are available. The CA File contains bibliographic information and index entries for over 6 million documents cited in *Chemical Abstracts* since 1967. The Registry File, for substance identification, contains more than 7 million substances with over 10 million names. The CAOLD File contains references to pre-1967 literature [*19*]. Other bibliographic data bases of interest to chemical engineers are based on *Engineering Index* (Engineering Information, Inc., NY) and services provided by private companies, such as Dialog Information Services (Lockheed Corporation), BRS Information Technologies (Latham, N.Y.), and System Development Corporation (Santa Monica, CA) [*20*].

Physical Property Data Bases

One of the characteristics of the chemical industry is the bewildering array of different chemicals that can be found in a typical process. Petroleum, for example, is a complex mixture of thousands of different chemical compounds, most of which are present in very small amounts. Proper design and operation of a plant require a knowledge of the physical properties of the important materials used in the plant—properties such as density, boiling point, heat capacity, and toxicity. To complicate matters, a change in plant operation may introduce new materials. The data base should have tabulations of reported experimental data, and there should be a facility for estimating properties at conditions for which data are not available. One way to find physical property information is to search bibliographic data bases, and, for commonly used materials, compilations of properties are available in handbooks and other special publications. The advantages of creating electronic physical property data bases are obvious.

Simulators used for process flowsheeting, such as ASPEN or PROCESS, contain built-in physical and thermodynamic property data bases and provide a facility for the user to enter data for materials that are not in the data base. Other data bases are available from the American Institute of Chemical Engineers through the Design Institute for Physical Property Data, with a data base containing 26 physical property constants and 13 sets of temperature-dependent property correlations for more than 400 pure compounds [*21*]; the Institution of Chemical Engineers (United Kingdom) with a data base for some 850 commercially important compounds, which is available on a microchip-based add-on module and which can be used as a peripheral for PCs [*22*]; and Technical Database Services, Inc. (NY), with three data bases, which can be obtained on magnetic tape or used on line.

MICROCOMPUTERS FOR INSTRUMENTATION

The microcomputer applications described above make it clear that there is a revolution in the ways that chemical engineers make calculations, design plants, and store information. However, there is another revolution occurring in the laboratory and in the chemical plant—the incorporation of microcomputers in the instruments used to measure chemical and physical properties, as well as in plant control systems.

Unfortunately, it is rarely possible to obtain a direct measurement of a desired physical property. For example, if a continuous measurement of the composition of the product from a distillation column is desired, the engineer may have to settle for measuring the temperature and the pressure and then calculating the composition from known vapor pressures of the materials. Of course, if time permits, samples can be sent to an analytical laboratory where a chemical analysis can be made by another indirect method such as gas chromotography. Even a fundamental quantity such as temperature is obtained indirectly by measuring a change in resistance of a sensor or the voltage generated by a thermocouple. The process of measuring a physical or chemical quantity therefore takes place in two steps: First, a sensor is used to measure some related quantity, and second, the measured quantity is converted to the desired quantity by calculations. Microcomputers are revolutionizing instrumentation in two ways: first, by providing greatly increased computational ability for individual instruments (sometimes called "intelligent" instruments), and second, by coordinating the measurements made by many different instruments.

The Intelligent Instrument

An intelligent instrument can apply complex data reduction techniques to experimental measurements because it has a built-in microcomputer, and today, almost any instrument costing more than a few thousand dollars has some degree of intelligence. The sophisticated data analysis required for Fourier transform spectroscopy, for example, can now be carried out automatically in the instrument, whereas several years ago, a mainframe computer would have been used. The development of the intelligent instrument has not only resulted in a better instrument but has opened up entirely new fields of instrumental analysis. One exciting new development is the use of a microcomputer to link together two different instrumental techniques in a single instrument. For example, the combination of gas chromatography and mass spectroscopy (GC–MS) in a single instrument gives two complimentary views of the same sample and, in essence, a new dimension of the analysis. It is not a very large extrapolation to predict that instruments using three or more different instrumental methods will become available in the future.

In addition to greatly improved data reduction and analysis, other important auxiliary functions can be carried out by a microcomputer incorporated in an instrument. These include simplification of instrument controls, self-calibration, automatic checks to make sure that the instrument settings are consistent, automatic selection of instrument ranges and other settings, and storing of instrument settings for specific applications.

The revolution in instrumentation made possible by the microcomputer has only begun and will continue well into the next decade, according to Hirschfeld [*23*], who predicts that the major advances will be the elimination of "dumb" instruments, design advances in sensors and multiple-measurement instruments, and the development of sophisticated instruments that can be used by nonexperts.

Networks for Data Interchange

A second major use of microcomputers is in instrumentation systems where data from a number of instruments must be collected and coordinated. In addition to computation ability, there must be an efficient means to inter-

change data rapidly among equipments. At the simplest level, a personal computer can be equipped with one of the many commercially available add-in boards [24], which can be installed directly in a personal computer. They are available at modest cost and can provide the following functions:

1. *Analog-to-digital (A/D) conversion.* The output from instrument sensors is usually a current or a voltage. This must be converted to digital form for use in the computer. The number of significant bits in the conversion (10 to 16) and the number of conversions per second (up to 80K) are important features, differentiating one board from another. Some boards provide simple signal processing, such as a programmable gain and compensation for thermocouple reference junction temperature or for nonlinearity of sensing devices.
2. *Multiplexing.* Multiple analog inputs can be serviced by one A/D converter by providing the ability to switch from one analog input to another. Typically, 8 to 16 inputs are provided, but as many as 256 can be accommodated on some boards.
3. *Digital-to-analog (D/A) conversion.* Numbers calculated in the computer can be converted to a voltage and provided as input to an instrument.
4. *Digital input/output (DIO).* Information from instruments in digital form can be input to the computer, and internally generated digital information can be provided to the instruments and display devices.
5. *A clock for control of sampling times.*

Boards can also be provided with the ability to generate interrupts and to use direct memory access, so that data acquisition can proceed without intervention from the microcomputer. Control of an add-in board requires good software. As a minimum, the computer must be able to issue commands, receive status information, and exchange data with the board. Software is usually provided by the board manufacturer or sold separately by software suppliers and can be referenced from a high-level language such as FORTRAN or BASIC. Complete software packages for acquiring data, analyzing it, making statistical tests, and preparing tables and plots are also available [25].

When data-gathering requirements can no longer be met by an add-in board, an external module or add-on board can be used. The add-on board will have its own power supply and is usually provided with additional computing power, relieving the microcomputer of some routine calculations. Better shielding can be provided to protect against noise when converting low-level signals from instruments such as thermocouples. The external board can be mounted remotely from the microcomputer, which simplifies wiring, because a direct connection must usually be provided between each sensor and the A/D converter.

However, with an external module it is necessary to establish a digital communication link to the computer. For short distances, a bus extender can be used, but this will not be satisfactory in noisy environments or for longer distances. One solution is to use the RS232 port, which is either standard equipment or an inexpensive accessory for microcomputers. This is a widely used serial protocol for communication between two devices. Speeds of 38K bits per second (approximately 3800 characters) and higher can be attained. If higher speeds are required or there are more than two devices, a parallel bus such as the IEEE-488 standard is indicated.

The IEEE-488 [26] standard was originally developed for Hewlett-Packard instruments and is also called the general-purpose interface bus (GPIB). It is well accepted by manufacturers, and most instruments either come with the interface or can be provided with one as an option. It uses 16 wires, 8 of which are used for parallel transfer of data. Up to 15 devices can be connected to the bus. One device is the controller, and all other devices are talkers, listeners, or both. Only one talker is permitted at a time. The controller determines which device is the talker and which are listeners and is able to pass control to another device. Transfer rates as high as 1Mb per second can be achieved.

MICROCOMPUTERS FOR CONTROL OF CHEMICAL PROCESSES

Process control applications have one important characteristic that distinguishes them from the applications discussed above—the overriding importance of real-time interaction with a process. The control system is responsible for keeping the process safe as well as profitable. The importance of safe operation can hardly be overemphasized in view of the possible disastrous consequences of a failure in plant safety such as the one that occurred in Bhopal, India. The control system must monitor process operation continually, make changes in plant operation, maintain a record of plant operation, and provide the information needed by human operators in a timely manner. Because microcomputers have the capability of doing these tasks better than was possible in the past, they are being widely incorporated in all new process control systems. However, it is absolutely essential that these functions be carried out reliably and that provision be made for equipment failure, human errors, and natural calamities. Before discussing how microcomputers are used in control systems, some of the techniques used to assure reliable operation will be treated.

Control System Reliability

Here, we consider primarily reliability of the control system itself rather than reliability of the complete process. One of the important functions of a control system is to detect unsafe plant operation and to take appropriate action. Special problems arise when the control system does not work properly. It goes without saying that a reliable control system must be constructed with the best components available, must be thoroughly tested, and must have a regular maintainance program. In addition, even the best systems must have an effective built-in error-detection and error-correction ability. The control system must be designed to check itself continually for errors.

Error Detection

1. Checks must be made to verify that the control system is carrying out its assigned tasks at the correct times. This is a consequence of the fact that the computer is controlling a process that operates inexorably in real-time. The computer has many tasks to perform, but some are more important than others, and some must be done at specific time intervals. It is quite possible that a key action could be omitted because of a software error or an unusual combination of process conditions. An independent check on whether a certain task

has been performed can be done with a "watchdog timer"—a counter that is connected to the system clock and is being decremented continually. Every time the task associated with the timer is executed, the timer is reset to its maximum value. If it reaches zero, it means that the task was not done in the maximum time allotted, and an error condition is established.

2. All transmission of data between elements of the control system should be done with internal error-checking techniques, such as parity checks and check sums.
3. The data being received from the process can be checked for reasonableness. Data that are clearly incorrect should generate an error condition. One way of checking key process data is by making redundant measurements—using two or more different sensors to detect the same temperature, for example. If the values do not check, an error condition is then indicated.
4. Regular checks of computer operation can be made by using test programs. The operation of instruments can be checked by using built-in reference standards, which are now available on newer, more intelligent instruments. Data transmission can be checked by sending data to another device and requesting the device to send it back.
5. The operation of key control elements can be verified by direct sensing. A critical control valve might be equipped with a sensor indicating valve position. The fact that the valve is operating properly can then be checked directly. The integrity of the power supply for the computer and instrumentation system is obviously of key importance and must be checked regularly.

Error Correction

After an error is detected, it must be corrected. Some of the actions that can be taken are detailed below.

1. Prompt acknowledgment of the error and recording of the error condition. If the error is minor, it may be sufficient to note the error in an error log. For more serious errors, higher supervisory levels should be warned. This usually requires a message to an operator console and may initiate a special warning such as an audible signal or a flashing light.
2. Not only is it highly desirable that the computer have the facility for detecting the error but it should also be capable of locating the reason for the error. This requires some diagnostic ability to be programmed into the computer, an ability that can be provided by what are called "expert systems" [*27*]. This is a promising new development for control system applications [*28*], and systems suitable for the chemical process industries are also becoming available [*29*].
3. If the error is occurring in some element of the control system, a spare or redundant piece of equipment can then be switched into operation. It is good practice to provide spare equipment for all important control functions. A particularly vulnerable item is the communication link between instruments and controllers, which could be destroyed in case of a fire, explosion, or other disaster, so an alternate communication channel should always be provided. If an error is indicated in a computer program or computer algorithm that is contained in random access memory (RAM), the program can then be rebooted from a disk or other permanent storage. The most critical programs are kept in read-only memory (ROM). A spare, completely independent power supply must be available in the case of a power failure.
4. Alternate control algorithms can be used where appropriate. For example, failure of a certain sensor might be compensated for by switching to a different control algorithm for the control loop in question. In a more drastic action such as an emergency shutdown, a completely different control algorithm for the entire plant might be initiated.

Control System Hierarchies

Process control systems are usually organized in a hierarchical manner, with specific functions being carried out at each level. The lowest levels control only a small portion of the plant. Each higher level coordinates the operation of lower levels until there is control of an entire plant or an entire company at the highest level. Low levels are designed to operate independently of higher levels, if necessary, so that control integrity can be maintained even during emergencies. A description of a five-level hierarchy follows [*30*]:

Level 0 *Sensors and actuators.* These are the devices that connect directly to the process.

Level 1 *Regulatory control.* Information from the sensors is used to generate settings for the actuators. The conventional proportional-integral-derivative (PID) control algorithms are usually used, although techniques such as cascade control, multivariable control, and feed forward are found in the newer, more advanced systems.

Level 2 *Supervisory control.* The coordination of lower-level controls for a complete production unit is accomplished at this level. This may include changing of control settings to achieve optimal operation of the entire unit. Plant-operating personnel will usually intervene in the control system at this level.

Level 3 *Facility management.* The integration of all production units in a complete production facility (e.g., a petroleum refinery) is carried out at this level. Because the product from one production unit is often the feed to another, it is necessary to coordinate the operation of all units.

Level 4 *Management information systems.* At the head of the hierarchy is the management system used by corporate management. Production schedules must be set up for each production facility based on customer demand, availability of raw materials, and many other factors.

Control System Computers

Minicomputers and mainframe computers are commonly used for control at levels 3 and 4. Special-purpose microcomputers are widely used at levels 1 and 2 because of the stringent reliability requirements and because of the fact that level 1 computers must interface with sensors and actuators. A single, powerful microprocessor-based, multiple-loop controller can accommodate up to 2000 input/output devices [*31*] (process sensors, actuators,

communication links, etc.). Two or more of them are called a "distributed control" system because each is dedicated to the control of a portion of a production unit.

The control system manufacturer usually markets a complete line of compatible sensors, actuators, and computers. This includes a communication system and protocol for interchange of data, called a data highway, which is often proprietary, because of lack of accepted standards. However, these same suppliers do not usually provide the larger computers needed for levels 3 and 4, and it will be necessary to provide a "gateway" between systems. Again, the lack of universally accepted standards for control system communication is a serious handicap to control system integration. One encouraging development is the good reception that the MAP (manufacturing automation protocol) [32] standard has received among suppliers and users. It was created by General Motors, based on existing standards developed by various international standards organizations and is primarily concerned with factory automation. The considerable purchasing power of Gereral Motors has ensured that most suppliers will support it. Because it was developed for factory automation, there has been some concern that it might be less satisfactory for continuous chemical processes. However, its wide acceptance will mean that control system manufacturers must, as a minimum, support a gateway based on MAP [33].

Control System Design

Control system design requires a knowledge of the dynamic behavior of a process—how process variables change in time with changes in plant operation. The equations describing the dynamic behavior are sets of simultaneous differential equations and algebraic equations and are more difficult to solve than the steady-state equations. Ideally, it should be possible to simulate the dynamic response of a system and test out various control methods. This is difficult to do. Although there are some general-purpose simulation programs available, for example, CSMP (Continuous System Modeling Program) [34], programs oriented to chemical process simulation are just beginning to be available [35]. Better simulation packages will permit better initial design of control systems and, even more important, as microcomputer capabilities expand, will permit the incorporation of dynamic simulations in the control system so that the effect of changes in process conditions can be accurately forecast and compensated for.

CHEMICAL ENGINEERING IN MICROCOMPUTER MANUFACTURE

It is evident from the preceding sections of this article that microcomputers are indispensable in chemical engineering practice. It is not so obvious, however, that chemical engineers are important in the manufacture of microcomputers, a subject that will be developed in this section. The heart of the microcomputer is the integrated circuit, or chip, which contains the equivalent of hundreds of thousands of circuit elements in an area 5 millimeters square. Spacing between adjacent elements can be of the order of 1 micron, far smaller than a human hair. The circuit elements and the connections between them are made using chemical reactions and chemical etching—exacting chemical processing steps, some of which must be carried out at high vacuum and high temperature and in "clean rooms" where all dust particles have been removed. A knowledge of reaction engineering, kinetics, thermodynamics,

and transport phenomena—all subjects in which chemical engineers are trained—is needed, and as a result, employment of chemical engineers in the electronic industries has risen sharply in the last few years. In 1981, about 3% of all offers to B.S. chemical engineers were from electronics companies. In 1984, this rose to about 12%. Larrabee describes the chemical engineering aspects of microelectronic manufacturing accurately and is the source of the description that follows [36].

The raw material for chip manufacture is metallurgical silicon with a purity of about 98%. This must first be purified to contain less than 1 ppb (part per billion) of impurity. The process involves reacting the silicon with hydrogen chloride at high temperature to produce trichlorosilane ($SiHCl3$) and halogenated impurities. The trichlorosilane is purified by distillation and then reduced with hydrogen to pure silicon at 1,100°C. Worldwide consumption of pure silicon is estimated at less than 5,000 metric tons, which makes it a small volume product for the chemical industry, but it has a very high value—perhaps $100,000 per ton.

A single pure crystal of silicon is next obtained by melting the silicon (over 1,400°C) and introducing a crystal seed, which is then drawn from the melt. During this process, very small amounts (typically 0.5 to 5 ppb) of impurities, such as arsenic, boron, or phosphorous, are added to give the silicon the desired semiconductor properties. The silicon crystal is large (5 or 6 inches in diameter and several inches long) and is next sawed into thin wafers, which are then polished and chemically cleaned in two steps to remove organic and inorganic surface contaminants. Each wafer is the basis for several hundred devices that are formed on the surface of the wafer by chemical reactions, diffusion, photolithography, and chemical etching.

There are many steps in the preparation of an integrated circuit. In a typical sequence for one such step, the surface of the wafer is oxidized in a furnace at 1,000°C to cover the surface with a thin insulating layer of silicon dioxide. Next a photosensitive polymer (resist) is deposited on the surface and is then exposed to light using a mask to create the desired pattern in the polymer. If a negative is used, the portion of the polymer exposed to light becomes less soluble and can be removed by a developer solvent. A portion of the silicon dioxide layer is thus exposed and is removed by chemical etching using, perhaps, an ammonium-fluoride-buffered hydrofluoric acid solution. The surface, which is covered with polymer, is not affected by the etching. The next step might be to add a dopant or impurity to the exposed silicon surface. This can be done by putting the wafer in a furnace and contacting the surface with the desired dopant in gaseous form. The dopant must diffuse to the surface of the wafer, adsorb on the surface, react on the surface, and then diffuse into the solid. After the desired degree of diffusion has taken place, the remaining silicon dioxide and polymer resist are removed chemically or mechanically, and another step is begun. After the last step, a layer of metal is deposited and then etched to form the connections between components. The wafer is then cut to separate the individual devices, which are then packaged.

It is evident even from the brief description above that the making of integrated circuits is a complex sequence of chemical operations, which are not different in principle from those used in the chemical industry, although the materials and operating conditions are unusual. An AIChE committee examining future research needs and opportunities in chemical engineering has suggested that the greatest contribution of the chemical engineer to the

development of microelectronic materials will be converting the batch operations used today [37] to continuous operation.

REFERENCES

1. *Chem. Eng.* p. 10 (August 5, 1985).
2. *1985 AIChE Applications Software Survey for Personal Computers*, American Institute of Chemical Engineers, New York, 1985.
3. *The Software Catalog*, Elsevier, New York, 1984.
4. *Engineering and Scientific Programs for IBM Personal Computers*, Engineer and Scientists Directory, IBM, Boca Raton, FL.
5. David J. Deutsch, *Microcomputer Programs for Chemical Engineers*, McGraw-Hill, New York, 1984.
6. *Chemical Engineering Software Guide*, CAE Consultants, Yonkers, NY.
7. R. W. H. Sargent, Process Systems Engineering: Challenges and Constraints in Computer Science and Technology, " in *Proceedings of the Second International Conference on Foundations of Computer-Aided Design*, CACHE Publications, Ann Arbor, MI, 1984, p. 23.
8. John D. Perkins, "Equation Oriented Flowsheeting," Ref. 7, p. 309.
9. F. M. Julian, "Flowsheets and Spreadsheets," *Chem. Eng. Prog.*, 35 (September 1985).
10. B. S. Gottfried and J. W. Tierney, "How to Use Your Microcomputer to Simulate a Coal Preparation Plant," *Coal Age*, 83 (November 1985).
11. C. G., Morris, W. D. Sim, and T. Vysniauskas, "An Interactive Approach to Process Simulation," *Chem. Eng. Prog.*, 41 (September 1985).
12. Nicholas Basta, "Computer Hardware Finding Niches in CPI Engineering-Design Work," *Chem. Eng.*, 14 (April 29, 1985).
13. V. E. Wright, "Microcomputer CAD Systems," *Chem. Eng.*, 14 (July 8, 1985).
14. R. E. Peters, "Integrated Computer-Aided Engineering," *Chem. Eng.*, 95 (May 13, 1985).
15. F. B. Canfield, "Computer-Aided Engineering in the Process Industry," *Chem. Eng. Prog.*, 31 (September 1985).
16. W. G. Beazley, "Transferring CAD Flowsheets, Part 1," *Chem. Eng.* 61 (October 28, 1985); "Transferring CAD Flowsheets, Part 2," *Chem. Eng.* 113 (December 9, 1985).
17. Keith Jones, "International Group Prepares Worldwide CAD/CAM Standard," *Mini-Micro Syst.* 44 (January 1986).
18. Peter Winter, and C. J. Angus, "The Database Frontier in Process Design," *Proceedings of the Second International Conference on Foundations of Computer-Aided Process Design*, CACHE, Austin, TX, 1984, p. 75.
19. Mo. S. Khan, Rod D. Pinkston, and David F. Zaye, "Chemical Engineering Information: What's Available and How to Get It," *Chem. Eng. Prog.* 82, 20 (1986).
20. Nicholas Basta, "Chemical Engineers Are Going On Line," *Chem. Eng.*, 16 (October 28, 1985).
21. B. A. Feay, T. E. Daubert, R. P. Danner, M. S. High, and C. L. Rhodes II, "Interactive Computer for DIPPR Data," *Chem. Eng. Prog.*, 80, 55 (1984).

22. *Chem. Eng.* 11 (June 10, 1985).
23. Tomas Hirschfeld, "Instrumentation in the Next Decade," *Science*, 30, 286, (1985).
24. Jesse Victor, "Add-in/Add-on Boards Extend Micros' Reach," *Mini-Micro Syst.*, 85 (October 1985).
25. Pamela S. Zurer, "Computers Gaining Firm Hold in Chemical Labs," *Chem. Eng. News*, 21 (August 19, 1985).
26. "IEEE Standard Digital Interface for Programmable Instrumentation," ANSI/IEEE Standard 488-1978, IEEE, Inc., New York, 1978.
27. Vaughn Voller and Brian Knight, "Expert Systems," *Chem. Eng.*, 93 (June 10, 1985).
28. William R. Nelson and James P. Jenkins, "Expert System for Operator Problem Solving in Process Control," *Chem. Eng. Prog.*, 81, 25 (December 1985).
29. "Process Controllers don 'Expert' Guises," *Chem. Eng.*, 14 (June 24, 1985).
30. Cecil L. Smith and Joachim Hirche, "Integrated Process Information and Control," *Chem. Eng.*, 113 (November 12, 1984).
31. G. J. Cordova, H. I. Hertanu, and G. T. Doyle, "Microprocessor-Based Distributed Control Systems," *Chem. Eng.*, 86 (January 21, 1985).
32. Robert Crowder, "The MAP Specification," 2d ed., *Control Eng.*, 22 (October 1985).
33. James J. McCarthy, "MAP's Impact on Process Plants," 2d ed., *Control Eng.*, 67 (October 1985).
34. Frank W. Speckhart and Walter L. Green, "A Guide to Using CSMP—The Continuous System Modeling Program," Prentice-Hall, Englewood Cliffs, NJ, (1976).
35. R. W. H. Sargent, Ref. 7, p 37.
36. Graydon B. Larrabee, "A Challenge to Chemical Engineers—Microelectronics," *Chem. Eng.*, 51 (June 10, 1985).
37. Joseph Haggin, "Chemical Engineering Frontiers: NRC Report Expresses Optimism," *Chem. Eng. News*, 14, (December 2, 1985).

JOHN W. TIERNEY

CHEMISTRY, ARTIFICIAL INTELLIGENCE APPLICATIONS

(see Artificial Intelligence Applications in Chemistry)

CIRCUITS, MICROPROCESSOR

1.0 OVERVIEW

The advent of inexpensive microprocessors and microcomputers has made it possible to experiment with different interconnections of multiple microcomputers to form different architecture for the parallel processing of information.

The nature of the architecture is studied using analyses based on such criteria as languages, type of programs, and so forth. The major driver in virtually all of these designs is speed. One difficulty is that for general mixes of programs to be executed, it is difficult to predetermine how the computer will perform. Many design choices that must be made in constructing the computer architecture also have an effect on performance. It is virtually impossible to evaluate these multiple alternatives once the computer is constructed. One difficulty in analyzing computer performance analytically is the different transient conditions that exist for different architecture.

Multiple microcomputers are typically interconnections of multiple von Neumann architecture. It has been suggested that the single most important metric in measuring system performance for von Neumann architectures is the number of bits passing the system bus [*1*]. Thus, the measure has the units of bits per second, a rate. It would be expected that an effective analysis tool would be one that could handle general functions of time.

One method of analysis of highly parallel computer structures is in the mapping of the graphic structure of the computer program onto a proposed or existing computer architecture.

A collection of interconnected microcomputers can be considered a circuit in the sense of electrical circuits where each microcomputer is simply a block of some type of lumped parameter, that is, the individual microcomputers would be analogous to the lumped parameters of inductance (L), capacitance (C), and resistance (R). If it is possible to establish such an analogy that can make use of the powerful tools of analysis and synthesis of circuit theory, it will mark a distinct advancement in the analysis and synthesis of computer architecture. A differential equation formulation of the problem would then permit a solution of both the transient and steady-state situations.

This article presents a discussion of certain factors involved in such an analogy. It does not contain an analogy or even suggest that a useful and consistent analogy exists. Instead, it suggests a way of looking at multiple interconnected microcomputers in the hope that one can eventually be found, such as in the case of mechanical systems [*2*] where the method of analysis has proved to be very powerful and useful.

Even if a consistent analogy cannot be found, the conceptual framework of thinking of the problem in this manner can be most effective when considering design alternatives and understanding the whys and hows of certain types of computer behavior.

1.1 THE CONTENTS

This article is divided into two parts. The first is a general discussion of the thinking of the form of an analogy; the second is one method of constructing microcomputer circuits for the solution of specific structures of computational problems.

1.2 INTRODUCTION

The definition of any circuit must, of necessity, contain references to some sort of path over which some entity can flow or move from one point to another, dependent on certain characteristics of the elements making up the circuit.

In general, a microprocessor is an integrated circuit with a package having some number of pins, processing power, speed, and so forth. The addition of memory, timing, input/output, and so forth, is accomplished by connecting (support) chips to the microprocessor through a simple set of connections or through a more general structure, making use of something termed a bus.

The interconnection of the microprocessor and support chips (a microcomputer) provides a circuit for electrical impulse that allows the exchange of binary information among the chips. From an electrical perspective, this is a circuit. From an informational perspective the microcomputer can be thought of as an element.

1.3 RLC* CIRCUITS

One of the most common types of circuits used by electrical engineers is the passive RLC circuit, consisting of resistors, inductors, and capacitors interconnected by conductors, typically wires. These circuits have an input and a function that produce an output.

Depending on the function and the form of the input, a number of circuit analytical techniques exist, ranging in difficulty from simple to complicated. As one views the history of RLC circuit analysis, the sophistication of the analysis techniques has been significantly increased by relatively few conceptual developments.

Much analysis is still done and taught on the basis of the simple parallel and series results of a few elements connected together. There are now more sophisticated analysis techniques, including computer automation of the techniques, but there was a time when the simple rules were the best available.

The current analysis of multiple microcomputers is at the same stage as the early RLC circuit analyses. We can understand simple economies of speed by paralleling two or more microprocessors (or microcomputers), or the bottleneck (analogous to impedance) of information of one microprocessor being forced to go through another when connected as a series element.

Much of our current circuit analysis of microprocessors is being aided by the work on computer networks. These are, in fact, computer circuits

*RLC indicates resistor (R; unit, ohm), inductor (L; unit, henry), and capacitor (C; unit, farad).

although their function or goal is more of a local element relying on the circuit simply for certain types of data, that is, the function of the circuit is to facilitate a limited function for the collective elements, for example, communication.

Electronic mail is becoming a novel and useful function for a circuit of computers where the tremendous power of the computer is only used on simple input/output functions. This is a result of the computer hardware being so cheap. In fact, this circuit relies almost totally on the existing telephone communication network for its global interconnection, although locally such things as local area networks (LANs) are in use and under study.

Part of the reason why microprocessor circuit analyses are still in a primitive stage is simply our conception of what a microprocessor is, that is, a small computer, an element unto itself. In time, this will change, and with it the effort on microprocessor circuit analysis will increase, and the techniques will become more sophisticated.

Another reason for the current state of the art of microprocessor circuits is the lack of definition of a function, that is, a problem to be solved. At this point, either our existing circuits are sufficient to solve our problems (or implement our functions) or we do not have the ability to perceive the problems that could be solved by sophisticated circuits.

But perhaps the most important reason, as alluded to previously, is the delay in recognizing the fact that hardware is now so cheap, there no longer exists a strong reason for requiring every microcomputer to be completely occupied all the time in doing its processing functions.

Still another reason is the lack of definition of the characteristics of the microprocessor or its electrical function, which are important to the functional implementation of the circuit.

For example, with RLC circuits, any characterizations are primarily in terms of voltage/current relationships, with the elements having resistance, inductance, and capacitance. There are other possibilities, such as charge and flux, that can be used by substituting other relationships into the resulting voltage/current relationships.

The simplicity of the RLC elements aided the analysis process because of the conservation laws that could be written for circuits leading to the early analysis developments, for example, conservation of charge, potential, and so forth.

On the other hand, because of their flexibility and capability, microprocessors are used for a multitude of properties. Thus, it is difficult to say exactly which characteristic(s) is important and should be investigated for use with circuit analysis of implementations.

We are making progress (although not directly) because of computer networks. At this point, it is important to point out that the two words "network" and "circuit" will be used for two distinct concepts. Networks refers to the heuristic interconnection or computers using such things as the telephone networks or LANs for the purposes relating to communications. Circuits refers to the interconnection of microprocessors (or computers) using any type of interconnection media, the goal being to solve a specific problem with the combination of the elements.

In a rather trivial sense, the computer networks solve a problem, that is, simplify communication. However, in the larger context, this is analogous to the interconnection of RLC elements because they conduct electricity.

1.4 AN ANALOGY

As an example of the type of function that can be characterized by a microprocessor circuit, consider delay. In an RLC circuit, the time for a capacitor to reach a certain steady-state voltage or an inductor to reach a certain steady-state current is frequently given by the time to reach 63% of its final value. When multiple elements are interconnected, the problem is complicated by many simultaneous relationships that must be satisfied.

Delay is a common characteristic of microprocessors due to the discrete nature of the clocking of the cycles of execution. At this point, the various factors are not important—simple that a delay does exist and can be realized. Thus, if an input (characteristically unimportant) is placed on any point of the circuit containing one or more microprocessors; there is a delay in that input or some function of that input reaching another point in the circuit. In the nontrivial case, this means passing through at least one microprocessor.

In a computer network, this is typically the time for a message to go from the sender to the receiver, a single point in the network; alternatively, in the broadcast mode of interconnection of elements, this means to all intended receivers. This time and its method of calculation are aided by straightforward mathematical relationships. In terms of system theory, this is an "open loop" type of situation.

Consider the computer network with an input point and an output point, where the output depends on all computers doing something to the input where each simultaneous output is observed at each element input once any delay time has passed.

It is important to note that the delay time mentioned above depends on and introduces a concept of memory that is different from the typical memory associated with a microprocessor, that is, the addressable locations for storing program, data, temporary results, and so forth. For this reason, program memory, data memory, and register memory are termed, with a qualifier on the word memory to distinguish a local hardware component(s) from the more general meaning of memory and its part in microprocessor circuits.

Consider a simple example where a single unit of information, data, or intelligence is input to a microprocessor circuit and passes through all microprocessors with each only providing a delay. The input to each microprocessor is not simply the first input observed but rather the accumulation over time of all inputs owing to possible differences of delay for various microprocessors.

The above example assumes a trivial input/output relationship of unity or identity simply accounting for delay. The analysis of this problem was first reported in 1975 [3]. The resulting analytical tool is a set of linear differential equations that can be conveniently arranged in matrix form.

One facet of the problem under consideration that must be mentioned at this point is the nature of continuous or discrete processes. It is not necessary for this work to go beyond some elementary concepts. The RLC circuits typically make use of certain physical relationships that can be expressed in a continuous form as follows:

$$v = iR \text{ or } v = L\frac{di}{dt} \text{ or } v = \frac{1}{C}\int_0^t i\,dt^{\dagger} \quad (1.1)$$

Because of the nature of the voltage (v) and current (i) relationships and the use of calculus to obtain solutions for equations, which can be expressed in terms of the various circuit configurations, the continuous nature of the process is important.

For the RLC circuit using the relationships above, the parameters of design for a given configuration are the R, L, and C of the relationships given in Equation 1.1. When selecting the actual circuit elements to construct a given circuit, the first criterion of choice is the values of R, L, and C. There are other factors, such as tolerance and working voltage, which satisfy constraints placed on the design by other considerations outside the design process.

The design procedures have greatly influenced our thinking by placing the framework of R, L, and C as the nominal variables to satisfy a specified circuit function. Questions, for example, of whether the C should be (I) MICA, (II) MYLAR, or (III) CERAMIC, whether it should be of (A) 100 WVDC, (B) 50 WVDC, or (C) 16 WVDC, whether the area of the plates should be (1) X, (2) Y, or (3) Z, or whether the leads should be (a) AXIAL or (b) RADIAL are normally outside the fundamental design procedure. There are methods of assisting the designer with the answers to these questions; however, the kernel of the design approach is typically outside of the questions.

The point of this discussion is to note the complex nature of the design process if all considerations are taken into account. The individual questions can be noted as being element specific as opposed to circuit specific.

At this point, the design process involving most microprocessor design considerations revolves about characteristics typically considered, such as (I) NMOS or (II) CMOS, (A) 2K or (B) 4K, ROM, (1) 2 MHz, (2) 4 MHz, or (3) 8 MHz, or (a) 8-bit, (b) 16-bit, or (c) 32-bit data buses.

Each of the indicated design considerations is analogous to the (I), (A), (1), (a) considerations with the RLC circuits. What is missing in the case of the microprocessors is the input/output relationships that permit the description of the microprocessor in terms of output as a function of input.

Certain problems, especially those involving real-time control, require specific functions to be implemented on microprocessors in real time, for example, fast Fourier transforms, which are used as a basis of comparison for both microprocessors and algorithms.

Certain architecture lends itself to solving problems in real time, such as digital filters implemented on modified Harvard architecture, permitting description as a signal processing function in contrast with a data processing function.

In RLC circuit analysis, it is normally assumed that the leads on the elements and the wires connecting the elements are to have no resistance, inductance, or capacitance, which means they have no effect on the voltage or current at any point in the circuit. In fact, this is not the case; but in comparison to other factors and within typical measurement capabilities, it represents a good approximation.

†The question of initial conditions would only complicate the problem at this point although an analogy exists. R indicates resistance in ohms, L indicates inductance in henries, and C indicates capacitance in farads.

With microprocessors, the analogy to the wire is the bus (serial or parallel) that connects the microprocessors into some form of circuit. Without the formalization of the microprocessor input/output characterizations, it is difficult to evaluate and/or disregard the effect on the quantities of interest, that is, the analogies to voltage and current.

Without getting into ideal elements at this point, it is possible to look at the analogies and complexities involved with microprocessor circuits. Rather than a single wire that provides a path for current flow, the data, information, or whatever characterization of the variable of interest flowing from microprocessor to microprocessor, is either transmitted (conducted) in serial or parallel form.

Consider first the parallel implementation that can be any number of bits depending on the precision of the data involved, that is, how many bits does it take to characterize a single unit of data or information. Any number of bits less then that required for a single unit characterization must involve some form of serialization of the transmitted variable.

For example, if the variable (data) unit is 8 bits, and four parallel conductors are used to connect microprocessors, then the 8 bits must be time sliced into 2 nibbles.

Although it is possible that the quantities analogous to voltage and current may not be described as functions of time, it is assumed to be highly unlikely. Thus, any serial implementation of interconnection will result in a nonideal connection between the microprocessors involved.

At this point, it is tempting to begin the establishment of an analogy. For example, wire size is a determinant of current-carrying capacity of the conductor. The number of parallel bit paths interconnecting microprocessors is the determinant of instantaneous data-carrying capability. Why not make a current/data analogy?

It might be possible to begin with such an analogy, but it would be premature. Without a definition of the variables, descriptors, functions, and so forth, the exercise is not a waste of time, but at this point, it is still pedagogical.

As an example of the type of analogy that would be useful, consider the circuit of Fig. 1.1.

There is an analogy that has been used for some time involving electrical and mechanical systems, as indicated in Table 1.1.

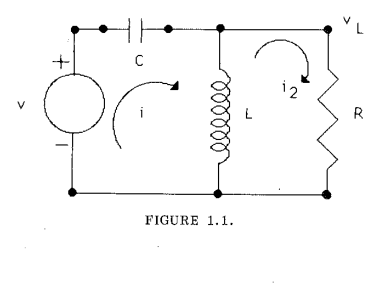


FIGURE 1.1.

TABLE 1.1 Electrical and Mechanical Systems Analogy

Electrical	Figure-Symbol	Mechanical
Voltage	v	Force
Current	i	Velocity
Flux linkage	∅	Displacement
Inductance	L	Mass
Resistance	R	Damping coefficient
Capacitance	C	Compliance

With such an analogy, it is possible to analyze a mechanical system as an electrical system and vice versa. The reader is encouraged to see the indicated references for details.

The primary use of such an example is to indicate that the same conceptual arguments can be used to establish an electrical/computing analogy that can take advantage of related developments.

Currently, an analogy that satisfies the completeness of the electrical/mechanical analogy does not exist, but one day it will.

As an example of the possibilities that exist, consider the indicated analogous terms indicated in Table 1.2. This does not suggest that this analogy is worthwhile, complete, or consistent, it is simply a useful conceptualization.

As a simple example, consider the circuit of Figure 1.1 and the analogy of Table 1.2.

The data v as an input to loop 1 must equal the data accumulated in the buffer (C) for future use plus the data used for processing (L). The same data, v_L, could be output through R.

Consider the circuit of Figure 1.2, where the data output from a processing element L must also pass through an output channel R, thus reducing the data rate, i_2. Potential in this sense could be the ratio of data in to data out, v_R.

The reason for introducing these simple concepts is to indicate that analogies for the conservation of energy, momentum, and so forth, could be the conservation of information or data.

TABLE 1.2 Electrical/Computing Analogy

Electrical	Figure-Symbol	Computing
Voltage	v	Complexity (bytes)
Current	i	Data (bytes per second)
Flux linkage	∅	
Inductance	L	Processing
Resistance	R	Output (dissipation)
Capacitance	C	Buffer (bytes)

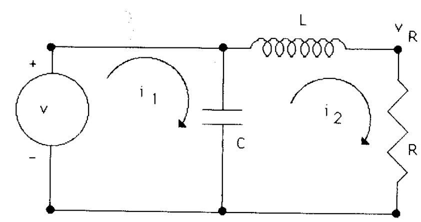


FIGURE 1.2.

2.1 INTRODUCTION

This section is concerned with certain microprocessor circuits that are in existence. Although the material is concerned with microprocessors, mention will be made of a circuit concept [4] that predates the microprocessor.

The development will use a number of circuit terms, although the material can be derived on the basis of heuristics and simple mathematics.

A circuit consists of nodes and branches indicating common points of connection of circuit elements. The concepts of parallel and serial are applied to the branches to indicate an ordering in time of current flow through the branches, that is, if branches A and B are in series, current will flow through A before it can flow through B. If A and B are in parallel, then current changes in A and B can occur simultaneously.

Consider the analogy of current flow and data flow. If A and B are in series, and A is "before" B, then the data from A must be available before B can operate on the data. There is a delay associated with any calculation or processing in A, for example, d(A).

For n elements in series, the time required for an output is

$$d(A_1) + d(A_2) + \ldots + d(A_n) \tag{2.1}$$

after the data are input to A_1.

If elements $B_1, B_2, \ldots, B_m$ are in parallel and connected to A_1 and A_2 as shown in Figure 2.2, the time delay for all of the data to get to the input of A_2 is

$$\text{DELAY} = \text{MAX}\ (d[B_1], d[B_2], \ldots, d[B_m]) \tag{2.2}$$

and the total circuit delay from 1 to 2 is

$$d(A_1) + \text{MAX}\ (d[B_1], d[B_2], \ldots, d[B_m]) + d(A_2) \tag{2.3}$$

The time delay for any branch d(B) is not something that typically can be observed in time by observing the intermediate changes in the data com-

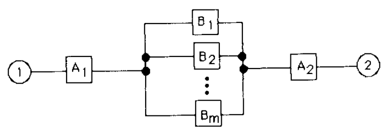


FIGURE 2.1.

pared with some initial/final value. A separate piece of information is required to note that a branch has completed its operation.

The completion may come from an addition bit, or it may be provided by a synchronous observation of some counter or interval detector. Both of these concepts will be considered.

2.2 ASYNCHRONOUS DIGITAL CIRCUITS

For the purpose of this section, an asynchronous completion signal will be assumed to be bundled with the parallel data lines; thus, a single line can be used to describe and draw the circuit.

With the completion signal, it is necessary to distinguish between fan out and fan in on the circuit nodes, both of which are shown in Figure 2.2.

The fan out physical connectors need only be a common wiring point. At the fan in node, it is necessary to use a device to note that all completion signals have been received. For those familiar with logic circuitry, this can be accomplished simply by a number of latches and an AND circuit.

Note that for data, no distinction is made for the number of parallel lines or bits. For example, in Figure 2.2(b), there may be three data paths into the node and one data path out where, for example, three data bytes may be required to determine a single data byte at the output of the branch.

Figure 2.3 shows the number of distinct wires in each branch notation for the case where 3 bytes are required to obtain 1 byte from branch element B.

Given the simple concepts presented to this point, it is possible to configure a circuit to process data where each branch performs some distinct operation(s) on the input data available at the time the completion signal is received.

Consider a circuit with an input A and a final output from branch Z, as shown in Figure 2.4.

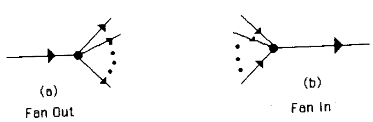


FIGURE 2.2.

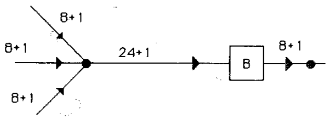


FIGURE 2.3.

As a conservative estimate, the completion signal of branch Z could be used as the signal to initiate branch A operation. However, branch A might be idle most of the time.

Consider grouping branches in such a way that the ith group of branches is noted as A_i and the delays are such that

$$d(A_1) = d(A_2) = \ldots = d(A_m) \tag{2.4}$$

Thus, the completion of A_i can be used as the initiation of branch A_{i+1} under the assumption that the data are available. This assumption is in order for what is to be presented, where it is assumed that the computing process is compute bound and not data bound.

The circuit under consideration is shown in Figure 2.5. In the following discussion, it will be assumed that the process delay for A_i is given by $d(A_i^*)$, with the small amount of time required to hold the data for A_{i+1} given by $\Delta(A_i)$ and, thus, the total delay given by $d(A_i)$, where

$$d(A_i) = d(A_i^*) + \Delta(A_i) \tag{2.5}$$

Once A_i has passed data to A_{i+1}, A_i can begin processing the new data. Thus, in time, the circuit of Figure 2.5 will have 1, 2, . . . , m elements of data in process.

For those digital processes where elements of data are to be continually processed, the above procedure allows the circuit to produce one result or final data element in time $d(A_i)$, although it takes m periods $d(A_i)$ until the first result is produced.

The above circuit is termed a "pipeline." Under asynchronous conditions, it is an "asynchronous pipeline."

Pipelines are valuable subsystems, especially for processing a stream or input data elements that may be continuous as in a real-time control system or the elements of a vector in traditional computational processes involving matrices.

The pipeline of Figure 2.5 has the following distinct components: (*a*) a function (what it does), F; (*b*) a data element width, W; (*c*) the number

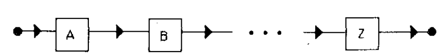


FIGURE 2.4.

FIGURE 2.5.

of elements, M; and (*d*) the delay per branch, D. Thus, the pipeline is characterized by (F,W,D,M).

The ith pipeline element is thus noted as P_i with

$$P_i = (F_i, W_i, D_i, M_i) \tag{2.6}$$

The pipeline allows a complex function, F, to be broken down into smaller operations, $A_1, A_2, \ldots, A_m$, requiring less complicated processors to implement each A_i and providing a parallel computational procedure to optimize the time to obtain a result.

2.3 MULTIPLE FUNCTIONS

Consider a computational procedure that can be described in statements of the substitution type of FORTRAN, as shown in Figure 2.6.

For the purpose of example, it is assumed that the process of interest is completely described by the statements of Figure 2.6.

The implementation of the circuit to obtain the result S is shown in Figure 2.7, where data inputs are assumed to be available as needed.

The circuit of Figure 2.7 looks very much like a pipeline, except that additional operations can be injected at various points in the circuit, that is, K, M, P, R, and T. The circuit can handle a continuous stream of data and requires six distinct processors.

Consider a problem where the number of data elements to be processed as inputs, H and I, are not continuous but limited to some number, ν. The number of data elements input at H and I is assumed to be such that any pair of elements will be separated by a time greater than that given by

$$2 \cdot D_1 + 2 \cdot D_2 + D_3 + D_4 \tag{2.7}$$

Under this assumption, when the first element of result O is obtained, the processor P_1 can be used to perform $O \oplus R$. The same will be true for $Q \oplus T$ giving the circuit implementation of Figure 2.8.

$G = H \oplus I$ (1)

$J = K \ominus G$ (2)

$L = J \odot M$ (3)

$O = P \otimes L$ (4)

$Q = O \oplus R$ (5)

$S = Q \ominus T$ (6)

FIGURE 2.6.

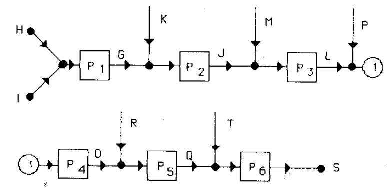


FIGURE 2.7.

Without going into detail, the primary differences between Figure 2.7 and Figure 2.8 are the fewer elements P_i required to implement the circuit and the necessity to time sequence the data being input to two of the nodes, that is, H and I and O, K, and T. In addition, the time to observe S must be aligned with respect to M.

The feature of Figure 2.8, where the same operations are repeated for some number of iterations, is very similar to the loop back concept employed in certain architectures.

The FORTRAN code that would provide the circuit of Figure 2.7 is given in Figure 2.9.

2.4 A GENERALIZATION

It is obvious that the circuit of Figure 2.8 is not going to work as a general "computer." However, referring to Figure 2.7, it is possible to generalize the indicated circuit as shown in Figure 2.10.

Each of the processors, $P_1, P_2, \ldots, P_n$, can be assumed to be identical, and thus each is capable of performing any of the indicated binary operations, if "o" is the operation.

Using these generalizations, the FORTRAN code of Figure 2.9 can be generalized, as shown in Figure 2.11.

This program can be arranged as shown in Figure 2.12.

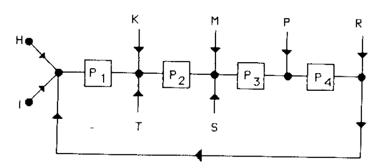


FIGURE 2.8.

```
      DO 10 N = 1, NTOTAL
      G(N) = H(N) + I(N)
      J(N) = K(N) - G(N)
      L(N) = J(N) * M(N)
      O(N) = P(N) / L(N)
      Q(N) = O(N) + R(N)
   10 S(N) = Q(N) - T(N)
```

FIGURE 2.9.

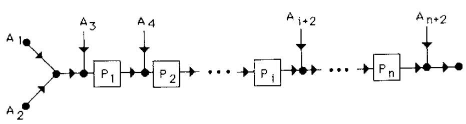


FIGURE 2.10.

```
      DO 10 I = 1, M
      X(1, I) = A(1, I) o A(2, I)
      X(2, I) = A(3, I) o X(1, I)
      X(3, I) = A(4, I) o X(2, I)
         .
         .
         .
   10 X(N, I) = A(N + 2, I) o X(N, I)
```

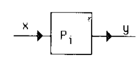
FIGURE 2.11.

```
      DO 10 I = 1, M
      X(1, I) = A(1, I) o A(2, I)
      DO 10 J = 1, N
   10 X(J + 1,I) = A(J + 2,I) o X(J, I)
```

FIGURE 2.12.

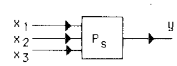
2.5 EXISTING PIPELINED CIRCUITS

The circuit of Figure 2.10 has particular significance in engineering. Consider a processor, P_i, that has the following input/output characteristic;



$$y = \int_0^t x \, dt \tag{2.8}$$

and a second processor type, P_s.



$$y = ax_1 + bx_2 + cx_3 \tag{2.9}$$

Given a differential equation of the form

$$\ddot{x} = a\dot{x} + bx + c \tag{2.10}$$

the circuit of Figure 2.13 generates a solution of the differential equation for discrete points in time, that is, the solution is of the form

$$(x[I] : I = 0, 1, 2, \ldots, N) \tag{2.11}$$

This solution is a vector and is of the form that is said to best suit the pipeline architecture.

The circuit of Figure 2.13 has a number of possible implementations. Each of the processing elements may be implemented by analog circuit elements, which are available on a typical analog computer [5]. The processing elements may also be implemented as specialized digital processors, as in a digital differential computer [6]. Alternatively, the processing elements may simple be microcomputers.

The vector structure of the differential equation can be seen from the discrete form of Eq. (2.13).

```
X2DOT(I) = A*   XDOT(I) + B*X(I) + C                    (2.12)
```

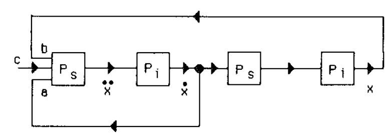


FIGURE 2.13.

```
DO 10 I = O, N

   CALL SUB1 (XDOT)
10 CALL SUB2(X)
```

FIGURE 2.14.

Alternatively, the differential equation may be mathematically transformed into a discrete form.

$$x(N + 2) = A^* x(N + 1) + B^* x(N) + C \qquad (2.13)$$

when, in general, $A \neq a$, $B \neq b$, and $C \neq c$. This equation can be implemented with the FORTRAN program shown in Figure 2.15.

This loop can be implemented with circuit shown in Figure 2.16. The delays in the circuit must be initialized with X(0) and X(1).

The circuit of Figure 2.16 can be implemented with a microprocessor for each block, giving us an example of both a pipeline and a parallel computation, that is, *A and *B can both be done at the same time.

2.6 ON AN ANALOGY

In the first part of this article, an analogy was discussed where a processing element might be characterized as a circuit element with some input/output relationship. Such an analogy would be straightforward in application.

Consider the situation of the previous section. Rather than make an element for element substitution, we have a computer configuration that from a data or signal standpoint, has an analogy in a differential equation, that is, given Figure 2.16, the equation is represented.

The differential equation has a direct analogy in terms of an electrical circuit. This circuit can be constructed (possibly, with certain key exceptions) and analyzed or simply analyzed mathematically.

2.7 SUMMARY

The advent of inexpensive microcomputers has made it possible to connect multiple microcomputers together in order to provide more powerful computing structures. Description and analysis of such configurations have typically followed the architectural conventions of digital computers. Such procedures lead to many difficulties in defining potential performance by analytical means. This is especially true in those conditions or situations that can be described as transient.

```
   DO 10 I = 1, N
10 X(I + 2) = A*X(I + 1) + B* X(I) + C
```

FIGURE 2.15.

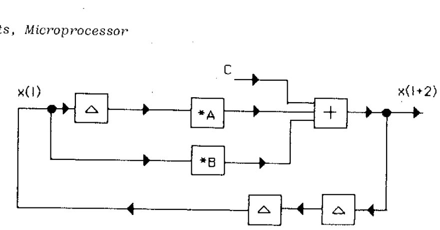


FIGURE 2.16.

Circuit analysis of electrical engineering provides a potential method of analysis. Even without a formal mathematical procedure, the conceptual framework can be helpful.

Certain traditional engineering methodologies, such as analog computer simultations of differential equations, have made use of parallel computing concepts. These concepts can be related to certain concepts in computer architecture.

This general framework of circuit analogies is believed to hold a great potential for analyzing parallel or distributed digital computer configurations.

REFERENCES

1. H. S. Stone et al., *Introduction to Computer Architecture*, Science Research Associates, Chicago, IL, 1980, pp. 530–531.
2. David K. Cheng, *Analysis of Linear Systems*, Addison-Wesley, Reading, MA, 1959.
3. M. H. Mickle and W. G. Vogt, "On the Dynamics of a Class of Optimum Routing Problems," *J. Franklin Inst.*, *300*(1), 7–23 (July 1975).
4. S. M. Ornstein, M. J. Stucki, and W. A. Clark, "A Functional Description of Macromodules," *AFIPS Proc.*, *30*, 337–356 (1967).
5. W. J. Karplus, "Analog Signals and Analog Data Processing," in *Encyclopedia of Computer Science and Technology*, (J. Belzer, A. G. Holzman, and A. Kent, eds.), Dekker, New York, 1975, Vol. 2, pp. 22–51.
6. R. E. H. Bywater, *Hardware/Software Design of Digital Systems*, Prentics-Hall, Englewood Cliffs, NJ, 1981, p. 4.

MARLIN H. MICKLE
WILLIAM G. VOGT

CIVIL ENGINEERING, MICROCOMPUTERS IN

INTRODUCTION

Microcomputers have gained substantial popularity among civil engineers. In addition to low cost, microcomputers appear to have a number of advantages over mainframe computers, which makes them appealing to civil engineering applications.

1. They provide a user-friendly programming environment. Users need not worry about complicated operating systems and JCL commands.
2. They lend themselves to interactive programming very effectively.
3. The problem of turn-around time associated with a batch mode environment is eliminated.
4. They provide inexpensive interactive graphic capabilities.
5. The problems associated with computing through modem, such as line interference, are eliminated.
6. They can often be expanded due to their open architecture.

STRUCTURAL ANALYSIS

Linear structural analysis of large structures with several hundred unknowns can be done on today's inexpensive microcomputers. Using the substructuring technique, dynamic analysis of a 42-story building with several thousand members was accomplished with about ten 20-minute runs on a CP/M 80 microcomputer [*1*]. Jonatowski and Koutsoubis [*2*] describe an interactive microcomputer program for static analysis of large space trusses. It can analyze a space truss with 3,500 members, 1,000 nodes, 6 load cases, and 20 load combinations. They report a significant gain in productivity compared with mainframe time-sharing methods. In time-sharing methods, a simple typing error would produce erroneous results, requiring hours of checking, and the smallest revision in geometry or loading would need considerable effort.

Brown and Kamat [*3*] present finite-element and boundary-element static structural analysis microcomputer programs for large-scale problems. A key feature of these program is the split BASIC and FORTRAN operation. They combined the superior input/output data manipulation and graphic capabilities of BASIC and fast computation speed of FORTRAN. They employed a compacted matrix storage scheme on the basis of the element-by-element (EBE) preconditioned conjugate gradient algorithm developed by Hughes *et al.* [*4*]. In this approach, only the upper triangular portion of the stiffness matrix of each element is stored, requiring much less storage than the in-core solution technique, utilizing the total element stiffness matrix with a compacted skyline structure. This is especially true for three-dimensional

problems whose stiffness matrix has a large mean bandwidth. They report the execution time for the static analysis of a 432-bar space truss with 288 degrees of freedom on an IBM PC equipped with an 8087 math coprocessor to be 127 seconds.

Peterson and Crouch [*5*] present a hybrid boundary element-finite-difference method for the stress analysis of underground mine layouts. The boundary element is used to determine the far-field stresses and displacements, and the finite difference method is used to determine the near-field stresses and displacements.

Wilson and Hoit [*6*] present a computer adaptive language for the development of structural analysis programs called CAL/SAP. It consists of a group of FORTRAN 77 subroutines intended to augment the standard FORTRAN language for the development of program modules in the general area of structural engineering. It utilizes the interactive feature of modern computers and data base management techniques available on modern operating systems. Three types of subroutines are implemented in CAL/SAP. They include a series of free-field input routines, a set of in-core data management subroutines that allow dynamic storage allocation with integer, real, and ASCII data, thus eliminating the paging problem on computers with virtual operating systems, and an out-of-core data management system allowing different programs to access the same data.

An important part of any structural analysis program is the formation and solution of linear equations. The two common approaches are the frontal method and the profile (active column) method. These approaches need the same number of floating-point operations when the equations are solved in the same order [*7–9*]. On microcomputers with slow floppy disk storage, the frontal approach does not seem to be suitable for analysis of large structures because the front must be subdivided and extensive disk access becomes inevitable. In the profile approach, on the other hand, the assembly of the structure stiffness matrix is uncoupled from the actual solution process. The structure stiffness matrix is assembled in a separate module in which element stiffness matrices are read in successively from secondary storage. The resulting structure stiffness matrix is in active column form in large blocks [*1*].

Microcomputer storage remains a problem for analysis of large structures. One common solution to this problem is substructuring. Chan *et al.* [*10*] describe a multilevel transfer substructuring method for the analysis of tall building structures using the transfer elimination technique.

Nonlinear finite element analysis of structures on microcomputers is practical for structures with localized nonlinearity and a limited number of nonlinear elements [*1*]. Mitri and Redwood [*11*] present a finite-element program for analysis of two-dimensional frames on microcomputers with material nonlinearity due to yielding and strain hardening. The material behavior is assumed bilinear, and the stress analysis in the inelastic range is carried out incrementally using the Newton–Raphson method and its modified version. They used a BASIC preprocessor for plotting the structure configuration in order to make an on-screen graphic check prior to program execution. Chidiac and Mirza [*12*] describe the development of microcomputer programs for thermal elastoplastic analysis of plates and shells.

Lawver and Saiidi [*13*] present a simplified model for inelastic lateral load analysis of short, symmetric highway bridges. The cyclic characteristics of nonlinear components are represented by a hysteresis model. They applied

this model to a five-span reinforced concrete multicell box girder bridge with a total length of 400 feet.

Using the kinematic approach with automatic generation of independent mechanisms, Adeli and Chyou [*14*] developed an efficient procedure and an interactive BASIC program for plastic analysis of irregular low-rise unbraced frames. The program can display/plot the frame configuration, including the loading, the bending moment diagram, and the failure mechanism. As an example, the failure mechanism of the frame shown in Figure 1 is shown in Figure 2.

Starbuck *et al.* [*15*] report the implementation of a microcomputer FORTRAN finite-element program for analysis of progressive fracture of reinforced concrete beams, using the principles of linear elastic fracture mechanics.

Wilson [*1*] lists the following desirable features for structural analysis programs to be developed on microcomputers:

They must run on both microcomputers and mainframe computers, that is, they must be portable. They must use efficient numerical techniques and new accurate finite elements. They must contain numerous error checks. They must be organized around a standard but flexible data base. They should use multilevel substructuring capability [*16*]. Wilson also suggests that they should be written in FORTRAN, primarily due to the large investment in existing software. In addition, the following desirable features may be added to the above list. They should employ interactive input/output. They should have an on-line help facility. They should employ graphic capabilities of microcomputers. This is best accomplished by employing separate pre- and postprocessors. This feature, however, works against the portability of the program because graphic primitives have yet to be standarized on various microcomputers.

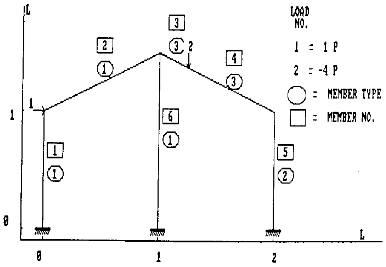


FIGURE 1

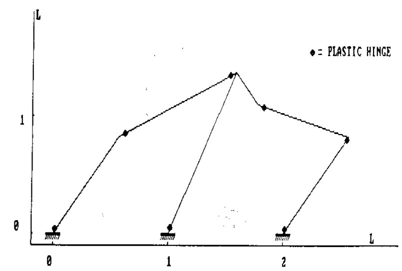


FIGURE 2

Considering the limited storage capacity of microcomputers, researchers have developed approximate methods that do not need large computer storage. Leung [*17*] presents an approximate method of analyzing three-dimensional tall buildings subjected to lateral loads by predetermining the deformation pattern at the nodes of a particular floor before loading.

COMPUTER-AIDED DESIGN

Computer-aided design (CAD) application in civil engineering is relatively new. One reason why CAD/CAM (computer-aided manufacturing) has not been utilized and developed in civil engineering as much as mechanical engineering or electronics is perhaps the uniqueness of civil engineering projects or "products." Although mass production has been the key word in manufacturing, major civil engineering products are mostly custom made. In other words, civil engineers must deal with the economy of scope, rather than economy of scale. Of course, there are also civil engineering products such as steel rolled sections, steel joists, precast structural elements, and industrial buildings that are mass produced.

Computer programs have not been used extensively in actual design of structures. A number of reasons may be cited for the lack of interest in conventional programs for design of structures [*18*].

1. In practical design cases, there are a large number of alternatives whose selection needs the judgement of the experienced human designer.
2. Design specifications usually cover the general situations and leave the less frequent cases to the judgment of the human designer.
3. Human designers use their previously gained experience in design of new structures.
4. A human designer usually visualizes and sketches different structural forms and configurations before making the preliminary design, stress analysis, and the final design.

5. Parts of design specifications and standards need interpretation by an experienced designer.
6. Design specifications change frequently, for example, every 3 years.
7. Design specifications are based on years of experience gained by researchers and practicing engineers and contain rules of thumb and heuristics that may not be readily implemented in traditional computer languages.
8. Design is an ill-defined and ill-structured problem, lacking a clearly defined goal and not quite amenable to algorithmic procedures.
9. Design is a creative process.

The first five problems can be alleviated by using the approach of "interactive design." In an interactive system, the designer is in charge, and the system serves as an assistant to him/her. Due to the open-endedness of the structural design problem, a general-purpose program for design of various types of structures does not seem to be feasible. Rather, CAD software should be restricted to one type or class of structures. Examples of interactive microcomputer-aided design (micro-CAD) of steel structures are presented by Adeli and Phan [*19, 20*], Adeli and Al-Rijleh [*21*], Adeli and Balasubramanyan [*22*], Adeli and Fiedorek [*23–25*], and Adeli and Chu [*26*].

Adeli and Fiedorek [*23–25*] present a microCAD system for interactive design of moment-resisting (type one) and simple (type two) connections in steel buildings made of standard rolled I-sections, on the basis of the American Institute of Steel Construction Specification (AISC, 1980). Connecting elements may be plates, angles, or T-sections. Connectors may be bolts or welds. The microCAD system can display/plot any isometric view of the connection plus three orthographic views, that is, front, side, and top views. An efficient priority list algorithm with back-face elimination is developed for the orthographic views. A five-test algorithm is developed for efficient drawing of the isometric views. Figure 3 shows an isometric view of a simple (type two) bolted angle connection. Figure 4 shows the

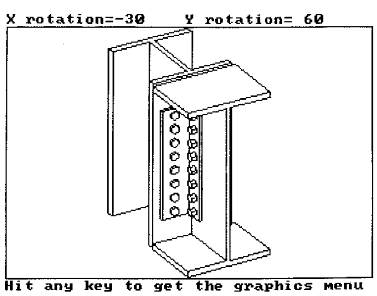


FIGURE 3

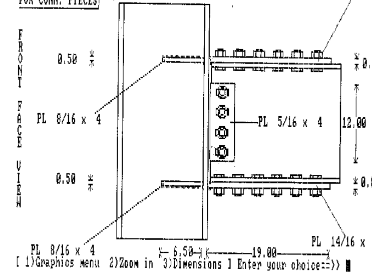


FIGURE 4

front-face view of a shop-welded and field-bolted, framed flange plate moment-resisting (type one) connection.

Adeli and Hawkins [*27*] present an interactive microcomputer-based graphics preprocessor for CAD of plane frame structures. The preprocessor allows various manipulations of the frame structure, such as zooming, panning, shrinking, expanding, node labeling, and element labeling. Figure 5 shows a sample output obtained from the preprocessor.

Chiang and Gergely [*28*] present an interactive BASIC program for the analysis and preliminary steel design of two-dimensional structures for the Macintosh personal computer, using the specific graphic capabilities of the machine, including the mouse, menus, and windows.

Few papers exist in the literature in the area of microcomputer-aided structural optimization. Nguyen and Hadley [*29*] used a simple version of the gradient projection method [*30*] for optimization of a 10-bar truss on an IBM PC. An algorithm for optimal plastic design of low-rise unbraced frames of general configuration is presented by Adeli and Mabrouk [*31*]. This algorithm is based on the static approach of limit analysis, without generating independent mechanisms. Using the kinematic approach with automatic generation of independent mechanisms, Adeli and Chyou [*32*] present an efficient procedure and an interactive BASIC microcomputer program for optimal plastic design of low-rise frames of general configuration. The procedure also finds the failure mechanism automatically. The interactive program can display the failure mechanism, as well as the frame configuration, including the loading and the bending moment diagram.

TRANSPORTATION ENGINEERING

Microcomputers have found a myriad of applications in traffic data collection, reduction, and utilization. In addition to ordinary general-purpose microcomputers, there are now various special-purpose microprocessor-based

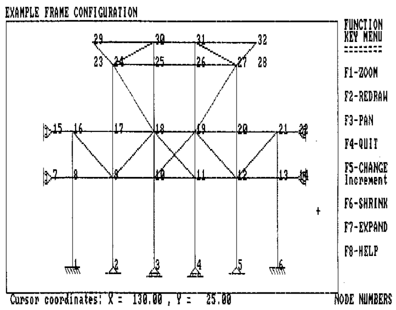


FIGURE 5

instrumentations for traffic counting, traffic flow studies, field inventory studies, and moving vehicle studies [*32*].

Microcomputer software available in the area of traffic analysis and management is reviewed by Skabardonis [*33*]. It includes programs for developing optimal signal timing plans along coordinated arterials and networks, estimating the capacity of isolated signalized intersections, and performing air quality modeling. He notes the lack of readily available software for the analysis of traffic operations on freeways and rural highways. Radwan and Sadegh [*35*] survey 12 microcomputer software packages available in the market, covering isolated intersections, arterial streets, and grid systems.

Khan [*36*] discusses microcomputers in highway system program management and reviews the software available in this area. Recently, the U.S. Department of Transportation [*37*] distributed two microcomputer programs, Pavement Management System (PMS) and Needs Inventory System (NIS), available on an IBM PC. Microcomputer software is rapidly increasing in the areas of traffic engineering, transportation planning, geometric design, and pavement design [*38*]. Khan [*36*] points out that there have been only modest attempts to develop microcomputer sofware in the area of highway system management.

Babey [*39*] presents nine software packages developed on a SUPER BRAIN microcomputer for the management of traffic operations. These programs address the following questions:

1. How to estimate current turning movements at an intersection, given an old turning movement count.
2. How to quantify the performance of a lane at a signalized intersection.
3. How to construct time—space diagrams for traffic signals in a linear network quickly.

4. How to summarize and analyze spot speed survey data.
5. How to find the desired storage length for a left- (or right-) turn bay at a signalized intersection.
6. How to find the goodness of fit between link volume predictions by an operational planning model and existing ground count information.
7. How to determine the best cycle time at an isolated traffic signal.
8. How to find a set of signal settings, given a cycle time at an isolated traffic signal.
9. How to determine the performance of all lanes at an isolated traffic signal.

A traffic data collection system is presented by Bell and Mohseni [*40*] for evaluating weight-in-motion, automatic vehicle classification, and automatic vehicle identification. King [*41*] presents a BASIC program for designing roadway lighting systems based on luminance considerations.

Taylor [*42*] and Skabardonis and Loubal [*43*] discuss some applications of microcomputer graphics in traffic engineering. Ramsey *et al.* [*44*] describe a microcomputer program for constructing space—time diagrams for analyzing train movements along railways. Michalopoulos and Lin [*44*] present a macroscopic freeway simulation program in which such phenomena as lane changing, merging, diverging, weaving, and friction effects are represented. The output of the program includes two- and three-dimensional plots of speed flow and density, visual representation of the freeway operation during the simulation, as well as description of the geometrics and demand pattern.

An innovative application of microcomputers is presented by Cox *et al.* [*46—47*]. They report a microprocessor-based pavement management system for the real-time monitoring of the cracks on the pavement surface to determine preventive and corrective maintenance using high-speed image processing. They used a multicamera video system mounted in a van and several 68020 microprocessors to handle and detect various cracking patterns while the vehicle is in motion at highway speeds.

Future applications of microcomputers in transportation engineering should include more extensive use of graphics in design of highway facilities, use of image-processing techniques for automatic vehicle detection and data analysis, and development of in-car route information and guidance system communicated by a centrally controlled traffic system for drivers.

CONSTRUCTION ENGINEERING AND MANAGEMENT

Applications of microcomputers in project management and construction estimating decision support systems, are presented by Moore [*48*]. Microcomputer-aided critical path management software is now being used in construction of large projects [*49*]. Cavaretta [*50*] discusses the use of microcomputers in the management of preconstruction activities, including design controls, negotiation of utility relocation agreements, environmental permitting, right-of-way acquisition, construction planning, maintenance of traffic sequencing, and contract package preparation for a large highway project with the assistance of spreadsheet, CPM scheduling, and commercial computer-aided drafting programs.

Ibbs [51] summarizes the conclusions of a recent workshop on future directions for computerized construction research. Four areas of research are discussed: (*a*) development, use, and maintenance of project-wide data base and communication systems, (*b*) knowledge-based systems, (*c*) simulation (analysis and design of site operation), and (*d*) robotics.

Rodriguez-Ramos [52] presents a construction layout model, using construction heuristics and microcomputer graphics. Karshenas [53] discusses microcomputer applications in earthwork volume measurement in construction projects.

Examples of automation in the construction industry can be found in the area of data acquistion, such as [54].

1. Video data acquisition.
2. Automated monitoring of construction quality control.
3. Automated monitoring of production rates and quantities.
4. Automated collection, storage, retrieval, preprocessing, and statistical classification of data.

Automation in the areas of process control and robotics has been quite limited so far, but potential applications are promising, including [54].

1. Microcomputer-based process control of fixed plants, such as concrete batch plants and precast concrete element fabrication plants.
2. Automation of mobile construction equipment, such as trucks, scrapers, loaders, shovels, graders, cranes, forklifts, pavement machines, and trenchers.
3. Fixed-based manipulators, also known as robot arm manipulators. Such an arm manipulator has been used in Japan for applying fireproof spray in building construction in a mobile platform [55].
4. Mobile robots and androids, including walking machines [56].
5. Microcomputer-based sensors for installation on structures under construction for monitoring and control.
6. Video image pattern recognition and image processing for monitoring operations.

WATER RESOURCES AND ENVIRONMENTAL ENGINEERING

Swayne *et al.* [57] and Fraser *et al.* [58] describe the development of a software in C language and Halo library of graphics for a user workstation for the analysis of acid rain data. The query language in the software is based on a pictorial representation of the geographical area under analysis. The product of the software system is intended to be an expert system for examining the relationship between terrain sensitivity indexes that assess susceptibility to acid deposition according to geologic and soil factors and resultant aquatic chemistry.

Mericase *et al.* [59] present a continuous real-time water quality monitoring and control system for soft-shell crab production in closed circulating aquaculture systems. Data collected from this system will be used to develop design criteria for the biological filters used in the crab shedding systems.

Chang and Liaw [60] outline an interactive water quality analysis and management program. It can find the dissolved oxygen profile of a river basin, perform a sensitivity analysis of the parameters that affect the profile, and obtain the least-squares treatment system for a river basin using the simplex algorithm. Use of microcomputers for optimizing the performance of waste treatment plants is discussed by Cochrane [61]. Jennings and Suresh [62] present an interactive microcomputer procedure for evaluating the relative risk of alternative hazardous waste management technologies, using the decision alternative ratio evaluation procedure. They are now employing fuzzy set analysis and decision theory to take into account user imprecision and uncertainty. Wei and Chen [63] present a microcomputer model for simulating the lime-soda softening of industrial cooling towers taking into account such factors as temperature, ion pairs, and ionic strength.

Petroski and Glebas [64] describe microcomputer digitization of river geometry for hydraulic modeling. Using microcomputer graphics, Kouwen and Harrington [65] describe the application of a pattern search optimization (hill climbing) technique to several hydrologic applications. Lam *et al.* [66] present a microcomputer-based model of the radionuclide spills in the wind-driven shore currents. Taylor *et al.* [67] discuss the development of computer-aided planning systems for water and other natural resources planning, using digital color-coded mapping. Microcomputer application in on-bottom stability analysis of fixed offshore platforms is reported by Jacobi [68]. Lindberg and Nielsen [69] present software for computer-aided analysis and design of urban sewer systems in Pascal. They are adding an expert module to the software for the identification of trouble spots during the simulation and for suggesting remedial measures.

Many hydrological applications of microcomputers can be found in the literature [70–72]. Miller *et al.* [70] describe a hydrologic floodplain model for estimating stream flows, water surface profiles, and water velocities resulting from a storm event. Blickwedehl [73] reports two microcomputer-controlled devices for taking measurements during the pump tests and to measure acquifer properties during slug tests. Graves *et al.* [74] report microcomputer programs for performing water well inventory, well log data retrieval, well construction details, well pumping test analysis, groundwater quality investigation, and acquifer simulations.

Aral [75] presents a finite-element Galerkin formulation for the solution of a steady-state groundwater seepage problem in a multilayer aquifer system for microcomputer environment.

GEOTECHNICAL ENGINEERING

Duplancic *et al.* [76] present a microcomputer system for automating the process of data collection, interpretation, and display in field and laboratory testing programs for geotechnical site characterization and soil exploration using commercial software. They used a spreadsheet program for data acquisition and data processing, a relational data base management system (dBASE II) [77] for data processing, and a drafting package for data display (alternative layouts of the boring logs). Edris *et al.* [78] describe the use of microcomputers for the collection of geotechnical construction control data associated with embankment dams using dBASE II [77].

LABORATORY APPLICATIONS

Microcomputers are being used increasingly in civil engineering laboratories. Selvadurai *et al.* [79] present a microcomputer-aided experimental system for the study of heat transfer processes in buffer regions of a nuclear waste disposal vault. Using the experimental facility, researchers study the thermal integrity of the buffer materials under moisture depletion conditions. The software developed for the system can display the rock mass temperature data as a two-dimensional view of the heat transfer process with color-designated temperature ranges.

Lam and Lam [80] describe the development of a microcomputer-controlled photogrammetric system that measures the photographic images of object points and computes their positions in object space coordinates. Zan [81] describes application of a microcomputer in the wind tunnel study of cable-stayed bridges. Tester and Gaskin [82] present a microcomputer system for monitoring and controlling a geotechnical frost susceptibility test. This test provides an index for classifying the frost susceptibility of soils for use in highway design.

Microcomputers are being used increasingly in geotechnical laboratories for both dynamic and static testing of soils and soil models. Several examples are given by Li *et al.* [83].

1. Data acquisition system for lateral pile loading to measure the magnitude and distribution of contact pressure along a pile.
2. Data acquisition system for pore water pressure and surface settlement measurements of an oil storage tank foundation model to be tested in the centrifuge.
3. Data acquisition and control system for free torsional vibration test to determine the shear modulus and damping ratio of soils.
4. Data acquisition and feedback control system for triaxial testing of soils under static or dynamic loading.

Wu and Khera [84] report a microcomputer-aided geotechnical triaxial test in which the graphics capability of microcomputers is used to plot the results. Figueroa and Yamamoto [85] present microcomputer data processing from dynamic tests on soil and rocks. They interfaced a desktop graphic tablet to a microcomputer for digitizing the resonant column device waveforms of decreasing amplitude to determine the Coulomb loss and hysteresis loops. Mould and Barker [86] describe a microcomputer-aided data acquisition and reduction system for studying the behavior of sheet pile interlocks under various loading conditions. The results of the experiments are being used to develop an analytical model to predict the behavior of cellular cofferdams subjected to construction, dewatering, and flooding loads.

Microcomputers have been used in a San Francisco Bay hydraulic model as a tide control system, as well as for automated data acquisition [87]. Jifeng and Zhengdong [88] also used a custom-made microcomputer system for the tidal model test of a river estuary regulation in China. Bonner and DePinto [89] developed a microcomputer-interfaced infrared laser nephelometric sinking meter to measure vertical transport rates of natural aquatic particles.

COMMERCIAL APPLICATION PROGRAMS

Spreadsheets

Electronic spreadsheets are among the most common commercial microcomputer application programs and have found popularity among civil engineers. More than 80 microcomputer spreadsheet programs can be found in the market [90]. Similar to a handmade accountant's spreadsheet, an electronic spreadsheet is a grid of columns and rows of cells. Each cell can hold and display a label, a number, or a formula. Any engineering computations that can be summarized in table form may be carried out by electronic spreadsheets. A spreadsheet program provides a general context for presenting logical and mathematical relationships.

Several attractive features may be cited for spreadsheets.

1. Ease of input. Any input item can be changed individually, making it very useful for examining various alternatives.
2. Windowing. Multiple windows allow the user to view various portions of input—output simultaneously.
3. Macros. The user can define menus. A sequence of menu selections can be stored in a macro. By invoking the macro name, the sequence will be executed.
4. Data base management. In addition to standard mathematical and statistical functions and financial formulas, spreadsheets have data base management capabilities, such as sorting and searching.
5. Graphics. Output can be presented in graphic forms. The graphic capability can also be used for creating design aids.
6. Portability. Work-sheet files may be transported from one microcomputer to another one with incompatible disk drives by using communications software [91].

Chu [90] discusses the application of spreadsheets in contract bid preparation and project budget control. He also describes the application of spreadsheets for evaluating liquefaction potentials of various soils in an earthquake. Duplancic *et al.* [76] used a spreadsheet program for data acquisition and processing in geotechnical site characterization and soil exploration.

Morris [92] shows the use of spreadsheet templates in construction scheduling and critical path analysis. Consuegra *et al.* [93] report application of spreadsheets in master drainage hydrological planning model. Use of spreadsheet in transportation planning (demand analysis) is discussed by Fung [94].

Structural analysis of simple structures with the help of spreadsheets is the topic of a paper by Lefter and Bergin [95]. Stiemer [96] presents application of spreadsheets in developing a self-paced courseware for teaching introductory structural steel design. Reinhorn and Kunnath [97] discuss the use of spreadsheets for design of reinforced concrete elements.

Data Base Management Systems

These programs are developed for efficient storage, retrieval, and manipulation of large quantities of data. The advantages of using data base management systems (or data base managers) include centralized control of

data, data integrity, data security, reduced redundancy of various data files, and data sharing among various users. Potential applications of data base managers in civil engineering have yet to be fully realized. There are now a number of commercial data base management systems such as dBASE III and Rbase. Some data base management systems are based only on one type of data structure, whereas others use a combination of data structures. The main data structures may be classified as hierarchical, network, and relational models.

The hierarchical model has a treelike structure composed of nodes and links. A node at a higher level is called parent, and nodes at a lower level are called children. No node can have more than one parent.

The network model is similar to the hierarchical model, with the difference that a node may be linked to any other node. Therefore, a node may have more than one parent. This model thus supports a more complicated data relationship.

Relational data bases are finding more applications in civil engineering due to their flexible data representation and retrieval capabilities. In a relational data base management system, data are represented as tables composed of rows and columns. Rows or tuples are records, and columns are fields occupied by attribute values. The column heading defines the name of the attribute domain. Wilson and Vogt [*98*] present an application of relational data bases in construction management, with emphasis on facilities management, and give an example for the renovation bid of a 24-story building. A relational data base manager has been used by Parker *et al.* [*99*] for management of data in wastewater treatment plants.

A relational data base management system, dBASE II [*77*], has been used by Duplancic *et al.* [*76*] to create a project-level data base management system for geotechnical data processing, by Elop and Heidebrecht [*100*] to develop a strong motion data base of earthquake records, by Chin [*101*] to develop a highway construction inspection management program, and by Sands and Hasit [*102*] in water quality and pollution control.

Data base managers can be used effectively to support CAD. In a complex CAD project a myriad of specialists and users are involved. Centralizing all data handling and collection increases the productivity and quality of the design. Engineering design data change frequently and are highly interdependent and partly in drawing form. Available commercial data base managers cannot handle these situations effectively because they are developed primarily with business applications in mind. A CAD data base manager should have the following characteristics:

1. It must be able to store various versions of a design due to the trial and error nature of design.
2. It must be able to support concurrent multiuser access.
3. It must be able to keep track of changes in design.
4. Interface of the CAD data base manager to other application programs should be generated easily.
5. It should have a memory management scheme.
6. It should have a query language suitable for interactive design.

Other desirable features are suggested by Murthy *et al.* [*103*].

Computer-Aided Drafting Systems

There is a large number of microcomputer-aided drafting systems in market. These programs are sometimes referred to as CAD or computer-aided design and drafting (CADD) systems, but they are, in fact, computer-aided drafting systems. A good review of 14 major microcomputer-aided drafting systems with a price range of \$300 to \$5800 is presented in the text by Smith and Teicholz [*104*]. They compare them in terms of drawing data base storage, two- or three-dimensional graphic manipulation, user interface, drawing details and symbol libraries, ability to create icons (pictograms), extensibility (whether the system can grow upward), system management issues (ability to perform a drawing backup and retrieval from disk, delete drawing, etc.), drawing capabilities, and intelligence embodied in the drafting system (automatic dimensioning in various units of measurements, text manipulation in combination with drawings, number of layers or sublayers that can be manipulated independently or collectively, dynamic zoom, pan, multiple drawing windows, and the ability to customize menus). Most of the more advanced microcomputer-aided drafting systems require specialized hardware such as high-resolution display monitors [*105*]. Four two-dimensional microcomputer-aided drafting systems under \$500 are reviewed by Milburn [*106*].

SINGLE VERSUS MULTIUSER MICROCOMPUTER SYSTEMS

Multiuser computers and operating systems so far have gained limited acceptance in the civil engineering community. Wilson [*1*] argues for superiority of single-user systems to multiuser systems for the following reasons:

1. Multiuser systems need a computer expert to run the system, to set up and maintain separate accounts, and to allocate computer resources.
2. A single-user system is easier to use because only one user is responsible for the selection of operating systems, languages, and file storage management. The execution time of a program does not depend on the number of users.
3. Several single-user systems are more reliable than one multiuser system. The failure of a multiuser system affects a large number of users.

Although single-user systems have dominated civil engineering offices so far, multiuser systems may find applications in situations where the allocation of microcomputer resources to various activities can be managed effectively.

MICROCOMPUTER LANGUAGES

The most popular microcomputer language for civil engineering applications seems to be BASIC. The next most popular language is FORTRAN. FORTRAN is favored for large-scale structural analysis programs [*1*]. Although a preferred language for scientific computations, FORTRAN is not an efficient language for screen input/output, graphics, and system level functions. Pascal and C languages have also been used for civil engineering applications to a lesser extent.

Lenocker and Young [107] discuss and compare processing speeds of microcomputer programs, which depend on

1. Type of microprocessor.
2. Clock rate of the microprocessor (usually between 4 and 10 megahertz).
3. Programming language.
4. Implementation of the language.
5. Amount of memory available to the program.
6. Size and access speed of disk storage.
7. Availability and capability of numerical coprocessor.

They conclude that C programs usually run the fastest, and BASIC programs run the slowest.

The above-mentioned programs are procedural languages designed basically for algorithmic numerical computations. For development of knowledge-based systems, however, declarative languages designed primarily for symbol manipulations, such as LISP and PROLOG, may be more suitable [18, 108]. The combination of a procedural language and a declarative language sometimes may be the best compromise [109].

FURTHER APPLICATIONS OF MICROCOMPUTERS IN CIVIL ENGINEERING

Computer Graphics

Until recently, popular microcomputers were basically text-based computers, with the exception of Macintosh with its built-in graphics package in ROM called QuickDraw. However, the new micros like Atari ST and Commodore Amiga are graphic based. Microcomputers with new architecture, such as the 32-bit RISC (reduced instruction set computer)-based IBM PC RT, provide new opportunities for developing graphics-based software. The lack of a graphics standard on microcomputers remains a problem. The application programmers sometimes may have to build or write drivers to create the necessary display commands. The primary hurdle in adopting a graphics standard appears to be the processing speed because the graphics computations put a heavy burden on the CPU. New microcomputers (e.g., Commodore Amiga), however, are using a special graphics microprocessor chip to drive the output devices such as CRT with high speed.

Two recently announced bit-mapped graphics chips will provide high graphics performance on microcomputers at low prices. They are the Intel 82786 and Texas Instruments TMS34010 [110]. The 82786 chip is a graphics coprocessor with functions designed to increase the speed of graphic interfaces, such as Microsoft's Windows and Digital Research's GEM. The TMS34010 Graphics Systems Processor is a 32-bit CMOS microprocessor that performs the functions of both a general-purpose CPU and a graphics processor.

Application of computer graphics in civil engineering has been limited mostly to the creation of drawings and has lagged behind other disciplines, for example, mechanical engineering for CAD [111]. Computer graphics should play a significant role in civil engineering during the coming years. Civil engineers can certainly take advantage of advances in computer technology, such as higher resolution and color of the display devices, faster CPUs, and improved secondary storage. One shortcoming of current microcomputers is their inability to store complicated drawings. By employing optical disks with 1 gigabyte (1,000,000,000 bytes) of memory, huge amounts of details can be stored [112].

Application of solid modeling in civil engineering has been quite limited. Due to extensive processing time and memory requirements, solid modeling has been limited so far to mainframe and superminicomputers. But, we should see migration of solid modeling to powerful microcomputers. Solid modeling will have applications in integrated CAD/CAM systems.

The most striking application of computer graphics is animation. The potential of computer animation in civil engineering design, research, and education has barely been explored [113]. Examples of application of animation in civil engineering are

1. Simulation of behavior of structures during earthquakes.
2. Simulation of wind flow around buildings.
3. Simulation of traffic in urban areas for studying queuing statistics, route selection, lane change, and signal coordination.
4. Animation of the sun rising on a high-rise building.
5. Simulation of storm surge activity on a lake [115].

Distributed Computing

As the number of microcomputers increases, the sharing of resources by different users will become a mecessity. Groups of civil engineers with various expertise located at distant physical facilities will be able to communicate and work together in a timely fashion through their individual desktop processors. Distributed computing should play a vital role in large civil engineering projects.

Distributed parallel processing when performed on multiuser single processors is referred to as multitasking (i.e., the ability to process several tasks concurrently). An example of such distributed task processing is presented by James and Unal [115] for integrating various hydrologic/hydraulic activities using local area networks. Nnaji [116] identifies several potential areas for multitasking in the field of water management and control, including multistage water allocation systems and flood forecasting at multiple sites along a river.

Networking

With the proliferation of microcomputers in the civil engineering community, the problem of information flow makes networking inevitable. Linking microcomputers to each other, to mainframe computers, and to pooled state-of-the-art peripherals, such as laser printers and plotters, should be a natural extension of the present usage. In addition to transfer of information, networking will make different phases of civil engineering projects, such as conceptual and preliminary design, analysis, design, drafting, detailing, estimating, fabrication, testing, construction, project management, inventory, accounting, and quality control, a single cohesive activity. Application of networking systems in civil engineering is expected to increase. This will be accompanied by higher transmission rates and the development of consistent protocols. In the somewhat more distant future, voice input/output will be an alternative way of communicating with the computer.

Artificial Intelligence and Expert Systems

Artificial intelligence (AI) is a branch of computer science concerned with making computers act more like human beings. Computer programs using AI techniques to assist people in solving difficult problems involving knowledge, heuristics, and decision making are called knowledge-based systems or expert systems. Applications of AI in civil engineering are starting to evolve [*108, 117*]. Until recently, AI research and expert systems were developed on large mainframe computers or dedicated AI machines. Microcomputers-based expert systems are now emerging. To facilitate the development of knowledge-based expert systems, expert system programming environments, or "shells," have recently been developed. These shells usually contain specific representation methods and inference mechanisms. Expert system shells available on microsystems have been reviewed by Wigan [*118*].

Several prototypical or conceptual models of expert systems for civil engineering applications have appeared in recent literature, some of which are microcomputer-based [*119*]. Ludvigsen *et al.* [*120*] review seven commercial expert system shells, three of which are suitable for microcomputers: Pascal-based EXPERT-EASE, Pascal-based INSIGHT II, and PROLOG-based M1. They point out a shortcoming of present expert system shells, that is, the inability to handle complex mathematical manipulations directly within the shell. Among the expert shells evaluated, only INSIGHT II can directly incorporate transcendental functions.

Levitt [*121*] reports the development of an expert system for evaluating the safety-related aspects of a contractor's organization and procedures on an IBM PC using the expert system shell DECIDING FACTOR [*122*]. O'Connor *et al.* [*123*] describe a microcomputer-based expert system for the analysis and evaluation of construction scheduling networks, using the expert system shell PERSONAL CONSULTANT PLUS. An attempt to use AI techniques in seismic risk analysis is given by Miyasato *et al.* [*124*]. Huang *et al.* [*125*] outline an expert system for fault diagnosis of hazardous waste incineration facilities, using the microcomputer-based expert system shell M1. They employed fuzzy probability computations [*126*] to take into account the imprecise nature of the knowledge of fault diagnosis. Milne [*127*] discusses development of an expert system for road curve design.

Gero and Balachandran [*109*] explore the application of knowledge engineering to pareto (multicriteria) optimization. They developed a prototype system on a SUN microsystem workstation. The domain-specific knowledge is represented in production (IF-THEN) rules and coded in LISP. The pattern-matching knowledge is written in PROLOG, and the optimization algorithm is developed in C.

One has to note that at present the expert system technology as applied to civil engineering problems is at an early stage, and the expert systems developed so far are basically experimental systems that should more appropriately be called pseudoexpert systems. But knowledge-based systems seem to have great potential in various civil engineering applications, such as design of roads, bridges, and buildings; management, maintenance, and diagnostic systems.

Smart Buildings

Central building microcomputer systems will find increasing applications. These systems will control the building environment (air-conditioning, air cleaning, central vacuum system, etc.), security, and communications.

REFERENCES

1. Wilson, E. L., "Structural Analysis on Microcomputers," in *Proceedings of the Second International Conference on Computing in Civil Engineering*, Hangzhou, China, 5–9 June 1985, Elsevier Science Publishers, New York, 1985, pp. 1–16.
2. Jonatowski, J. J., and D. Koutsoubis, "Microcomputer Engineering of Large Space Structures," in *Proceedings of the Third Conference on Computing in Civil Engineering*, San Diego, California, April 2–6, 1984, American Society of Civil Engineers, New York, 1984, pp. 498–505.
3. Brown, S. A., and M. P. Kamat, "Structural Analysis of Large Scale Problems using a Microcomputer," in *Proceedings of the Second National Conference on Microcomputers in Civil Engineering*, Orlando, Florida, October, 30–November 1, 1984, pp. 302–306.
4. Hughes, T. J. R., J., Winget, I. Levit, and T. E. Tezdauyar, "New Alternating Direction Procedures in Finite Element Analysis Based on EBE Approximate Factorization," in *Computer Methods for Nonlinear Solids and Structural Mechanics*, (S. Alturi and N. Perrone, eds.), ASME, AMD, 1983, Vol. 54, pp. 75–109.
5. Petersen, D. L., and S. L. Crouch, "Stress Analysis—A Coupled Boundary-Element/Finite-Difference Method," *Byte*, 11(7), 219–230 (1986).
6. Wilson, E. L., and M. I. Hoit, "A Computer Adaptive Language for the Development of Structural Analysis Programs," *Compt. Struct.*, 19(3), 321–338 (1984).
7. Wilson, E. L., and H. H. Dovey, "Solution or Reduction of Equilibrium Equations for Large Complex Structural Systems," *Adv. Eng. Software*, 1(1) (1978).
8. Taylor, R. L., E. L. Wilson, and S. J. Sackett, "Direct Equations by Frontal and Variable Band, Active Column Methods," in *Proceedings of the Nonlinear Finite Element Analysis in Structural Mechanics*, Europe–U.S. Workshop, Ruhr University, W. Germany, July 28–30 1980.
9. Hoit, M. I., and E. L. Wilson, "An Equation Numbering Algorithm Based on a Minimum Front Criteria," *Compt. Struct.*, 16(1–4), 225–239 (1983).
10. Chan, H. C., J. S. Kuang, and H. G. Li, "Multilevel Transfer Substructuring Analysis of Tall Buildings," *Proceedings of the First Canadian Conference on Computer Applications in Civil Engineering-Microcomputers*, McMaster University, Hamilton, Ontario, Canada, May 21–22, 1986, Vol. 1, pp. 189–200.
11. Mitri, H. S., and R. G. Redwood, "NAF2D: A Finite Element Program for Nonlinear Analysis of Frames in 2-Dimensions," in *Proceedings of the First Canadian Conference on Computer Applications in Civil Engineering—Microcomputers*, McMaster University, Hamilton, Ontario, Canada, May 21–22, 1986, Vol. 1, pp. 368–382.
12. Chidiac, S. E., and F. A. Mirza, "Thermal-Elasto-Plastic Analysis of Plates and Shells Using Microcomputers," in *Proceedings of the First Canadian Conference on Computer Applications in Civil Engineering-Microcomputers*, McMaster University, Hamilton, Ontario, Canada, May 21–22, 1986, Vol. 1, pp. 325–340.
13. Lawver, R. A., and M. Saiidi, "Inelastic Static Analysis of Laterally-Loaded Bridges on a Low-Cost Microcomputer," in *Proceedings of the Second National Conference on Microcomputers in Civil Engineering*, Orlando, Florida, October 30–November 1, 1984, pp. 267–271.

46. Cox, G. D. Curphey, D. Fronek, and J. Wilson, "Remote Video Sensing of Highway Pavements at Road Speeds Using the Motorola 68020 Microprocessor," *Microcomput. Civil Eng.*, 1(1), pp. 1–13 (1986).
47. G. Cox, D. Fronek, and R. Merrill, "Pavement Management System with Real Time Microprocessor-Based Computation," *Microcomput. Civil Eng.*, 1(2), pp. 95–105 (1986).
48. Moore, W. C., ed. *Small Computers in Construction*, ASCE, New York, 1984.
49. Robinson, R., "Building with Software," *Civil Eng.* 78–80 (May 1986).
50. Cavaretta, C. F., "Pre-Construction Project Controls with Microcomputers," in *Proceedings of the Third National Conference on Microcomputers in Civil Engineering*, Orlando, Florida, November 4–6, 1985, pp. 161–165.
51. Ibbs, C. W., "Future Directions for Computerized Construction Research," in *Proceedings of the Third National Conference on Microcomputers in Civil Engineering*, Orlando, Florida, November 4–6, 1985, pp. 239–246.
52. Rodriguez-Ramos, W., "On Construction Layout Modeling and Microcomputer Heuristics," in *Proceedings of the Second National Conference on Microcomputers in Civil Engineering*, Orlando, Florida, October 30–November 1, 1984, pp. 119–123.
53. Karshenas, S., "Microcomputer Applications in Earthwork Volume Measurement," in *Proceedings of the Second National Conference on Microcomputers in Civil Engineering*, Orlando, Florida, October 30–November 1, 1984, pp. 87–89.
54. Paulson, B. C., "Control of Construction Equipment Processes," in *Microcomputer Applications within the Urban Transportation Environment* (M. D. Abkowitz and R. Stammer, Jr., eds.), ASCE, New York, 1985, pp. 729–738.
55. Yoshida, T., *et al.* "Development of a Spray Robot for Fireproof Cover Work," in *Proceedings of the Workshop Conference on Robotics in Construction* (D. Sangrey, ed.), Pittsburgh, PA, June 17–20, 1984.
56. Weinstein, M. B. *Android Design*, Hayden Book Co., Rochelle Park, NJ, 1981.
57. Swayne, D. A., A. S. Fraser, D. C. L. Lam, and L. White, "Microcomputer Analysis of Aquatic Effects due to Acid Precipitation: Part I-Development of User Software," in *Proceedings of the First Canadian Conference on Computer Applications in Civil Engineering—Microcomputers*, McMaster University, Hamilton, Ontario, Canada, May 21–22, 1986, Vol. 2, pp. 15–21.
58. Fraser, A. S., D. C. L. Lam, D. A. Swayne, and L. White "Microcomputer Analysis of Aquatic Effects due to Acid Precipitation: Part II-Application of Eastern Canada Data," in *Proceedings of the First Canadian Conference on Computer Applications in Civil Engineering-Microcomputers*, McMaster University, Hamilton, Ontario, Canada, May 21–22, 1986, Vol. 2, pp. 22–33.
59. Mericas, C. E., D. G. Burden, and R. F. Malone, "A Microcomputer-Based Monitoring and Control System for Water Quality Management in an Experimental Aquaculture Facility," in *Proceedings of the Third National Conference on Microcomputers in Civil Engineering*, Orlando, Florida, November 4–6, 1985, pp. 64–68.

60. Chang, S. Y., and S. L. Liaw, "Interactive Programs in Water Quality Analysis and Management," in *Proceedings of the Third National Conference on Microcomputers in Civil Engineering*, Orlando, Florida, November 4–6, 1985, pp. 69–73.
61. Cochrane, J. J., "Optimizing Treatment Plant Performance Utilizing Microcomputers," in *Proceedings of the Third National Conference on Microcomputers in Civil Engineering*, Orlando, Florida, November 4–6, 1985, pp. 123–126.
62. Jennings, A. A., and P. Suresh, "Interactive Risk Analysis for Hazardous Waste Disposal Alterntives," in *Proceedings of the Second National Conference on Microcomputers in Civil Engineering*, Orlando, Florida, October 30–November 1, 1984, pp. 190–104.
63. Wei, I. W., and J. C. Y. Chen, "Modeling the Lime-Soda Softening of Cooling Waters by a Microcomputer," in *Proceedings of the Third National Conference on Microcomputers in Civil Engineering*, Orlando, Florida, November 4–6, 1985, pp. 127–131.
64. Petroski, M. E., and T. Glebas, "Microcomputer Digitizing of River Geometry for Hydraulic Modeling," in *Proceedings of the Third National Conference on Microcomputers in Civil Engineering*, Orlando, Florida, November 4–6, 1985, pp. 132–136.
65. Kouwen, N., and R. A. Harrington, "Automatic Estimation of Model Parameters," in *Proceedings of the First Canadian Conference on Computer Applications in Civil Engineering—Microcomputers*, McMaster University, Hamilton, Ontario, Canada, May 21–22, 1986, Vol. 1, pp. 443–457.
66. Lam, D. C. L., D. A. Swayne, M. Middleton, and M. Walma, "A Microcomputer-Based Model Radionuclide Spills and Discharge Plumes," in *Proceedings of the First Canadian Conference on Computer Applications in Civil Engineering—Microcomputers*, McMaster University, Hamilton, Ontario, Canada, May 21–22, 1986, Vol. 1, pp. 458–469.
67. Taylor, M., P. French, and D. P. Loucks, "Digital Color Mapping and Resource Planning," in *Proceedings of the First National Conference on Microcomputers in Civil Engineering*, Orlando, Florida, November 1–3, 1983, pp. 138–142.
68. Jacobi, J. G., "Microcomputer Applications in On-bottom Stability Analyses of Fixed Offshore Platforms," in *Proceedings of the Second National Conference on Microcomputers in Civil Engineering*, Orlando, FL, October 30–November 1, 1984, pp.
69. Lindberg, S., and J. B. Nielsen, "Modelling of Urban Storm Sewer Systems," in *Proceedings of the First International Conference on Applications of Artificial Intelligence in Engineering Problems*, (D. Sriram and R. Adey, eds.), Southampton University, United Kingdom, April, 1986, Vol. 2, pp. 687–696.
70. Miller, A. C., S. N. Kerr, W. P. James, and M. R. Jourdan, "MILHY: A Microcomputer Floodplain Model," in *Proceedings of the Second National Conference on Microcomputers in Civil Engineering*, Orlando, Florida, October 30–November 1, 1984, pp. 206–209.
71. Lam, A., P. Wisner, and J. Sabourin, "Development of a Multi-level Package of Stormwater Management Models," in *Proceedings of the Second National Conference on Microcomputers in Civil Engineering*, Orlando, Florida, October 30–November 1, 1984, pp. 210–219.

72. Patry, G. G., "NONLIN II—A Nonlinear Hydrologic Model for Microcomputers," in *Proceedings of the Second National Conference on Microcomputers in Civil Engineering*, Orlando, Florida, October 30–November 1, 1984, pp. 220–223.
73. Blickwedehl, R. R., "Application of Microprocessors to Geohydrologic Measurements," in *Proceedings of the First National Conference on Microcomputers in Civil Engineering*, Orlando, Florida, November 1–3, 1983, pp. 193–197.
74. Graves, B. J., P. Hall and J. Achilles, "The National Water Well Association's Ground Water Software Centre," in *Computer Applications in Water Resources*, (H. C. Torno, ed.), New York, 1985, pp. 1351–1360.
75. Aral, N. M., "A Regional Multilayered Aquifer Model for Microcomputers," *Microcomput. Civil Eng.*, 1(1), 68–78 (1986).
76. N. Duplancic, D. R. Rehak, and P. P. Christiano, "A Personal-Computer-Based Scheme for Geotechnical Data Processing and Display," in *Proceedings of the Second National Conference on Microcomputers in Civil Engineering*, Orlando, Florida, October 30–November 1, 1984, pp. 9–18.
77. Ashton-Tate, *dBASEII Assembly Language Relational Database Management Systems*, Culver City, CA, 1983.
78. Edris, E. V., Jr., W. E. Strohm, Jr., and L. A. Mann, "Use of Microcomputers for the Collection of Geotechnical Construction Control Data Associated with Embankment Dams," in *Proceedings of the Second National Conference on Microcomputers in Civil Engineering*, Orlando, Florida, October 30–November 1, 1984, pp. 136–140.
79. Selvadurai, A. P. S., K. C. McMartin and S. Conley, "A Computer-Aided Experimental Technique for the Study of Heat Transfer Processes in Buffer Regions of a Nuclear Waste Disposal Vault," in *Proceedings of the First Canadian Conference on Computer Applications in Civil Engineering—Microcomputers*, McMaster University, Hamilton, Ontario, Canada, May 21–22, 1986, Vol. 2, pp. 212–228.
80. Lam, L. C. H., and R. P. Lam, "Microcomputer Hardware and Software for Photogrammetric Applications in Civil Engineering," in *Proceedings of the First Canadian Conference on Computer Applications in Civil Engineering—Microcomputers*, McMaster University, Hamilton, Ontario, Canada, May 21–22, 1986, Vol. 2, pp. 199–211.
81. Zan, S. J., "Measurements of Cable-Stayed Bridge Dynamics Using an IBM PC," in *Proceedings of the First Canadian Conference on Computer Applications in Civil Engineering—Microcomputers*, McMaster University, Hamilton, Ontario, Canada, May 21–22, 1986, Vol. 2, pp. 229–240.
82. Tester, R. E., and P. N. Gaskin, "Controlled Freezing and Thawing of Soil Using a Microcomputer System," in *Proceedings of the First Canadian Conference on Computer Applications in Civil Engineering—Microcomputers*, McMaster University, Hamilton, Ontario, Canada, May 21–22, 1986, Vol. 2, pp. 325–337.
83. Li, X. S., C. K. Shen, and C. K. Chan, "Microcomputers in Geotechnical Engineering Experiments," in *Proceedings of the First National Conference on Microcomputers in Civil Engineering*, Orlando, Florida, November 1–3, 1983, pp. 138–142.
84. Wu, Y. H., and R. P. Khera, "Microcomputers in Geotechnical Testing," in *Proceedings of the Second National Conference on Microcomputers in Civil Engineering*, Orlando, Florida, October 30–November 1, 1984, pp. 141–144.

85. Figueroa, J. L., and T. Yamamoto, "Microcomputer Data Processing from Dynamic Tests on Soil and Rocks," in *Proceedings of the Third National Conference on Microcomputers in Civil Engineering*, Orlando, Florida, November 4–6, 1985, pp. 35–38.
86. Mould, K. M., and R. M. Barker, "Acquisition and Reduction of Experimental Data Using an IBM PC," in *Proceedings of the Third National Conference on Microcomputers in Civil Engineering*, Orlando, Florida, November 4–6, 1985, pp. 234–238.
87. Morton, M. R., and G. W. Domurat, "San Francisco Bay Hydraulic Model Data Acquisition and Control System," in *Proceedings of the Second National Conference on Microcomputers in Civil Engineering*, Orlando, Florida, October 30–November 1, 1984, pp. 253–257.
88. Jifeng, X., and W. Zhengdong, "The Application of Microcomputers in Estuarine Tidal Model Tests," in *Proceedings of the Second International Conference on Computing in Civil Engineering*, Hangzhou, China, 5–9 June 1985, Elsevier Science Publishers, New York, 1985, pp. 466–476.
89. Bonner, J. S., and J. V. DePinto, "A Computer Controlled Device to Measure Vertical Transport Rates of Aquatic Particles," in *Proceedings of the First National Conference on Microcomputers in Civil Engineering*, Orlando, Florida, November 1–3, 1983, pp. 181–185.
90. Chu, L. L., "Electronic Spreadsheets as an Engineer's Tool," in *Proceedings of the Second International Conference on Computing in Civil Engineering*, Hangzhou, China, 5–9 June 1985, Elsevier Science Publishers, New York, 1985, pp. 602–610.
91. Young, C. J., and J. L. Starnes, "Programming Financial Spreadsheets for Structural Member Design," in *Proceedings of the First Canadian Conference on Computer Applications in Civil Engineering—Microcomputers*, McMaster University, Hamilton, Ontario, Canada, May 21–22, 1986, Vol. 2, pp. 139–150.
92. Morris, D., "Critical Path Scheduling Using Spreadsheet Concepts," in *Proceedings of the Second National Conference on Microcomputers in Civil Engineering*, Orlando, Florida, October 30–November 1, 1984, pp. 77–81.
93. Consuegra, D., B. Morse, and P. Wisner, "Utilisation de LOTUS 1-2-3 pour l'Amelioration de la Gestion des Donnes d'un Modele Hydrologique," in *Proceedings of the First Canadian Conference on Computer Applications in Civil Engineering-Microcomputers*, McMaster University, Hamilton, Ontario, Canada, May 21–22, 1986, Vol. 1, pp. 470–485.
94. Fung, Y. H., "Use of LOTUS 1-2-3 for Transportation Planning," in *Microcomputer Applications within the Urban Transportation Environment*, (M. D. Abkowitz and R. Stammer, Jr., eds.), ASCE, New York, 1985, pp. 636–644.
95. Lefter, J. and T. Bergin, "Structural Analysis by Spreadsheets," *Civil Eng.*, 76–77 (May 1986).
96. Stiemer, S. F., "Microcomputers in Teaching: Steel Design with Spreadsheets," *Microcomput. Civil Eng.*, 1(2) (1986).
97. Reinhorn, A. M., and S. K. Kunnath, "Use of Spreadsheets for Design of Reinforced Concrete Elements," in *Proceedings of the First Canadian Conference on Computer Applications in Civil Engineering—Microcomputers*, McMaster University, Hamilton, Ontario, Canada, May 21–22, 1986, Vol. 1, pp. 16–27.

98. Wilson, J. L., and R. J. Vogt, "An Application of Relational Databases in Construction Management," in *Proceedings of the First Canadian Conference on Computer Applications in Civil Engineering—Microcomputers*, McMaster University, Hamilton, Ontario, Canada, May 21–22, 1986, Vol. 1, pp. 78–91.
99. Parker, D. G., H. G. Bray, and S. C. Parker, "Use of Relational Database Management Systems in Water Polution Control," in *Proceedings of the Second National Conference on Microcomputers in Civil Engineering*, Orlando, Florida, October 30–November 1, 1984, pp. 195–197.
100. Elop, S. A., and A. C. Heidebrecht, "The Development and Application of a Microcomputer Strong Motion Database for Earthquake Engineering Research and Design," in *Proceedings of the First Canadian Conference on Computer Applications in Civil Engineering—Microcomputers*, McMaster University, Hamilton, Ontario, Canada, May 21–22, 1986, Vol. 2, pp. 310–324.
101. Chin, S. M., "Highway Construction Inspection Management Using a Microcomputer," in *Microcomputer Applications within the Urban Transportation Environment* (M. D. Abkowitz and R. Stammer, Jr., eds.), ASCE, New York, 1985, pp. 739–748.
102. Sands, R., and Y. Hasit, "Database Management in Water Polution Control," in *Proceedings of the First National Conference on Microcomputers in Civil Engineering*, Orlando, Florida, November 1–3, 1983, pp. 177–180.
103. Murthy, S., Y. K. Shyy, and J. S. Arora, "MIDAS: Management of Information for Design and Analysis of System", *Proceedings of the 26th SDM Conference*, American Institute of Aeronautics and Astronautics, New York, pp. 85–95 (1985).
104. Smith, D., and E. Teicholz, *PC CADD: A Buyer's Guide*, Graphic Systems, Inc., Cambridge, MA, 1985.
105. Bellinghall, L., "The MicroCAD Explosion," *Comput. Graphics World*, 23–28 (March 1986).
106. Milburn, K., "Two-D Drafting for Under $500?" *Comput. Graphics World*, 33–38 (March 1986).
107. Lenocker, W. T., and C. Y. M. Young, "Speed Comparisons of Microcomputer Programs," in *Proceedings of the Third Conference on Computing in Civil Engineering*, San Diego, California, April 2–6, 1984, ASCE, New York, pp. 414–428.
108. Kostem, C. N., and M. L. Maher, eds., *Expert Systems in Civil Engineering*, ASCE, New York, 1986.
109. Gero, J. S., and M. Balachandran, "Knowledge and Design Decision Processes," in *Proceedings of the First International Conference on Applications of Artificial Intelligence in Engineering Problems*, (D. Sriram and R. Adey, eds.), Southampton University, United Kingdom, April, 1986, Vol. 1, pp. 343–352.
110. Stock, R., and B. Robertson, "New Chips Unleash Super Graphics," *Comput. Graphics World*, 24–32 (June 1986).
111. Flachsbart, B. B., "Reflections on the Impact of Computer Science on Engineering," in *Proceedings of the Ninth Conference on Electronic Computations*, Birmingham, Alabama, February 23–26, pp. 14–27, American Society of Civil Engineers, New York, 1986.
112. Malloy, R., "A Round-up of Optical Disk Drives," *Byte*, 11(5), 215–224 (1986).
113. Adeli, H., "Animation on Microcomputers," *Int. J. Civil Eng., Practicing Design Eng.*, 4(12) 1029–1042 (1985).

14. Adeli, H., and H. Chyou, "Plastic Analysis of Irregular Frames on Microcomputers," *Comp. Struct.*, *23*(2), 233–240 (1986).
15. Starbuck, J. M., W. E. Wolfe, and R. S. Sandhu, "Progressive Fracture in Concrete Using a Microcomputer," in *Proceedings of the First Canadian Conference on Computer Applications in Civil Engineering—Microcomputers*, McMaster University, Hamilton, Ontario, Canada, May 21–22, 1986, Vol. 1, pp. 142–154.
16. Wilson, E. L., and E. P. Bayo, "Use of Special Ritz Vectors in Dynamic Substructure Analysis," *J. Struct. Eng.*, *112*(8), 1944–1954 (1986).
17. Leung, A. Y. T., "Microcomputer Analysis of Three-Dimensional Tall Buildings," *Comp. Struct.*, *21*(4), 639–661 (1985).
18. Adeli, H., and Y. Paek, "Computer-Aided Design of Structures Using LISP," *Comput. Struct.*, *22*(6), 939–956 (1986).
19. Adeli, H., and K. Phan, "Interactive Computer-Aided Design of Non-Hybrid and Hybrid Steel Plate Girders," *J. Computer. Struct.*, 22(3), 267–289 (1986).
20. Adeli, H., and K. Phan, "Interactive Computer-Aided Load and Resistance Factor Design of Plate Girders," *Comput. Struct.*, 23(4), 509–534 (1986).
21. Adeli, H., and M. M. Al-Rigleh, "Computer-Aided Design of Trusses Using Turbo Pascal", *Microcomput. Civil Eng.*, 2(2), pp. 101–116 (1987).
22. Adeli, H., and K. V. Balasubramanyam, "Interactive Microcomputer-Aided Design of Circular Suspension Cable Roofs," *Comput. Struct.*, 23(6), pp. 837–844 (1986).
23. Adeli, H., and J. Fiedorek, " A MICROCAD System for Design of Steel Connections—Program Structures and Graphic Algorithms," *Comp. Struct.*, 24(2), pp. 281–294 (1986).
24. Adeli, H., and J. Fiedorek, "A MICROCAD System for Design of Steel Connections—Applications," *Comput. Struct.*, 24(3), pp. 361–274 (1986).
25. Adeli, H., and J. Fiedorek, "Microcomputer-Aided Design and Drafting of Moment-Resisting Connections in Steel Buildings," *Microcomput. Civil Eng.*, 1(1), pp. 32–44 (1986).
26. Adeli, H., and H. R. Chu, "Interactive Microcomputer-Aided Load and Resistance Factor Design of Frames," *Comput. in Civil Eng.*, 2(1), 1987.
27. Adeli, H., and D. Hawkins, "A Graphics Preprocessor for Computer-Aided Design and Drafting of Frame Structures", *Microcomput. Civil Eng.*, 1(2), pp. 107–120 (1986).
28. Chiang, K., and P. Gergely, "Interactive Structural Analysis and Steel Design on the Macintosh," in *Proceedings of the Ninth Conference on Electronic Computations*, University of Alabama, Birmingham, February 23–26, 1986, pp. 570–578.
29. Nguyen, D. T., and P. K. Hadely, "Microcomputers in Structural Optimization," in *Proceedings of the First National Conference on Microcomputers in Civil Engineering*, Orlando, Florida, November 1–3, 1983, pp. 30–34.
30. Haug, E. J., and J. S. Arora, *Applied Optimal Design*, John Wiley, New York, 1979.
31. Adeli, H., and N. Mabrouk, "Optimum Plastic Design of Unbraced Frames of Irregular Configuration," *Int. J. Solids Struct.*, 22(10), pp. 1117–1128 (1986).
32. Adeli, H., and H. Chyou, "Microcomputer-Aided Optimal Plastic Design of Frames," *J. Comput. Civil Eng.*, 1(1) pp. 20–34 (1987).

33. Courage, K. G., "Field Data Collection for Traffic Studies," in *Microcomputer Applications within the Urban Transportation Environment*, (M. D. Abkowitz and R. Stammer, Jr., eds.), ASCE, New York, 1985, pp. 240–247.
34. Skabardonis, A., "Use of Microcomputers in Transportation: Potentials, Limitations, and Future Trends," in *Microcomputer Applications within the Urban Transportation Environment* (M. D. Abkowitz and R. Stammer, Jr., eds.), ASCE, New York, 1985, pp. 240–247.
35. Radwan, A. E. and A. Sadegh, "Assessment of Software for Traffic Signal Analysis," in *Microcomputer Applications within the Urban Transportation Environment* (M. D. Abkowitz and R. Stammer, Jr., eds.), ASCE, New York, 1985, pp. 371–382.
36. Khan, A., "Microcomputers in Highway System Program Management: Software Requirements," in *Proceedings of the Third National Conference on Microcomputers in Civil Engineering*, Orlando, Florida, November 4–6, 1985, pp. 189–193.
37. U.S. Department of Transportation, *Road MAHP—Microcomputer Applications in Highway Projects*, Transportation Systems Center, U.S. Department of Transportation, Cambridge, MA, 1985.
38. U.S. Department of Transportation, *Microcomputers in Transportation: Software and Source Book*, U.S. Department of Transportation, UMTA/FHWA Technical Assistance Program, Cambridge, MA, 1984.
39. Babey, G. M., "Microcomputer Applications in the Traffic Operations Environment," in *Proceedings of the Second International Conference on Computing in Civil Engineering*, Hangzhou, China, 5–9 June 1985, (Elsevier Science Publishers), New York, 1985, pp. 331–349.
40. Bell, C. A., and A. E. Mohseni, "Processing and Presentation of Traffic Data from Oregon's Automatic Vehicle Monitoring System," in *Proceedings of the First Canadian Conference on Computer Applications in Civil Engineering-Microcomputers*, McMaster University, Hamilton, Ontario, Canada, May 21–22, 1986, Vol. 2, pp. 266–282.
41. King, L. E., "LUMSIM: A Microcomputer Program for Designing Roadway Lighting Systems," in *Proceedings of the Second National Conference on Microcomputers in Civil Engineering*, Orlando, Florida, October 30–November 1, 1984, pp. 119–123.
42. Taylor, M. A. P., "Microcomputers and Interactive Graphics: A New Communications Medium for Transportation Engineering," in *Proceedings of the First Candian Conference on Computer Applications in Civil Engineering—Microcomputers*, McMaster University, Hamilton, Ontario, Canada, May 21–22, 1986, Vol. 2, pp. 383–397.
43. Skabardonis, A., and P. S. Loubal, "Applications of Computer Graphics in Traffic Control," in *Microcomputer Applications within the Urban Transportation Environment* (M. D. Abkowitz and R. Stammer, Jr., eds.), ASCE, New York, 1985, pp. 342–350.
44. Ramsey, G. R. S., B. G. Hutchinson, and L. R. Rilett, "MICRORAIL: A Straight Line Railway Capacity Analysis Model," in *Proceedings of the First Canadian Conference on Computer Applications in Civil Engineering—Microcomputers*, McMaster University, Hamilton, Ontario, Canada, May 21–22, 1986, Vol. 2, PP. 426–437.
45. Michalopoulos, P. G., and J. Lin, "A Freeway Simulation Program for Microcomputers," in *Microcomputer Applications within the Urban Transportation Environment*, (M. D. Abkowitz and R. Stammer Jr., ed.), ASCE, New York, 1985, pp. 330–341.

114. Libicki, C. M., and K. W. Bedford, "Computer Animation of Storm Surge Predictions", *J. of Hydraulic Eng.*, 111(2) 284–299 (1985).
115. James, W., A. Unal, "Hydrologic Work Group System Using Distributed Task Processing," in *Computer Applications in Water Resources*, (H. C. Torno, ed.), ASCE, New York, 1985, pp. 1313–1321.
116. Nnaji, S., "Setup and Usage of Multitasking Microcomputers for Water Management Problem Solving," in *Computer Applications in Water Resources*, (H. C. Torno, ed.), ASCE, New York, 1985, pp. 1303–1312.
117. Adeli, H., "Artificial Intelligence in Structural Engineering," *Eng. Anal.*, 3(3), pp. 154–160 (1986).
118. Wigan, M. R., "Engineering Tools for Building Knowledge-Based Systems on Microsystems," *Microcomput. Civil Eng.*, 1(1), pp. 52–68 (1986).
119. Godfrey, K. A., Jr., "Expert Systems Enter the Market Place," *Civil Eng.*, 70–73 (May 1986).
120. Ludvigson, P. J., W. J. Grenney, D. Dyreson, and J. M. Ferrara, "Expert System Tools for Civil Engineering Applications," in *Expert Systems in Civil Engineering* (C. N. Kostem, and M. L. Maher eds.), ASCE, New York, 1986, pp. 18–29.
121. Levitt, R. E., "Howsafe: A Microcomputer-Based Expert System to Evaluate the Safety of a Construction Firm," in *Expert Systems in Civil Engineering* (C. N. Kostem and M. L. Maher, eds.), American Society of Civil Engineers, New York, 1986, pp. 55–66.
122. Campbell, A. N., and S. Fitzgerrell, *The Deciding Factor User Manual*, Power-Up Software, San Mateo, CA, 1985.
123. O'Connor, M. J., J. M. De la Garza, and C. W. Ibbs, "An Expert System for Construction Schedule Analysis," in *Expert Systems in Civil Engineering* (C. N. Kostem and M. L. Maher eds.), American Society of Civil Engineers, New York, 1986, pp. 67–77.
124. Miyasato, G. H., W. M. Dong, R. E. Levitt, A. C. Boissonnade, and H. C. Shah, "Seismic Risk Analysis System," in *Expert Systems in Civil Engineering* (C. N. Kostem and M. L. Maher, eds.), American Society of Civil Engineers, New York, 1986, pp. 121–132.
125. Huang, Y. W., S. Shenoi, A. P. Mathews, F. S. Lai, and L. T. Fan, "Fault Diagnosis of Hazardous Waste Incineration Facilities Using a Fuzzy Expert System," in *Expert Systems in Civil Engineering* (C. N. Kostem and M. L. Maher, eds.), ASCE, New York, 1986, pp. 145–158.
126. Zadeh, L., "Fuzzy Sets," *Info. Control*, 8, pp. 338–353 (1965).
127. Milne, P. H., "An Expert System for Road Curve Design and Setting-out," in *Proceedings of the First International Conference on Applications of Artificial Intelligence in Engineering Problems* (D. Sriram and R. Adey, eds.), Southampton University, United Kingdom, April, 1986, Vol. 2, pp. 733–743.

HOJJAT ADELI

CLIENT-CENTERED INFORMATION PROCESSING

INTRODUCTION

Self-direct information processing (SD-IP) by individuals is reviewed in relation to the information environment within which American society is now rapidly expanding. Scholars have cited various sources as evidence of the initiation of the information revolution, and numerous "celebrations" [*1*] have occurred that extol the very real benefits to be obtained. But for the masses of average people in the population, any real life evidence of personal and individual life style applicability does not seem very prevalent.

Unfortunately, such a gap between the aroused expectations of benefit among great numbers in the population and deliverable consumer products "that work" all too often fosters a Luddite backlash. In the current entrepreneurial environment, it is hoped that some popularly viable deliverable product would soon "make good" and remedy the limitations of such "volkswagen" efforts in a people's computer, as those of Apple, Leading Edge, and other products.

For the purpose of this article, two documents have often been recognized as instrumental in focusing attention on the many phenomena that have been associated with information consumer advocacy. The market, institutional, and professional (to some extent) aspects of the new commercial enterprise surrounding IP were initially considered to be delineated by Machlup [*2*]. This work provided extensive documentation for the new social environment that has arisen since mid-century, even though Machlup [*3*] was later to question the validity of the many cults surrounding the new information society.

The human IP aspects of the information phenomenon were carefully analyzed by Havelock [*4*]. Taking a sociological and sociopsychological perspective, Havelock developed the macro model of knowledge production, dissemination, and utilization (KPDU). To some extent, Havelock realized the vision expressed by Weaver [*5*] for an explication of the social and psychological implications of information theory. But more significantly, Havelock articulated the need for a consumer advocacy approach to information deliverables [*4*, Chapter 11].

This preliminary departure was advanced in the work, *Putting Knowledge to Use* [*6*], which analyzed and synthesized the literature about knowledge dissemination and social change. In more recent years, work in this area continues to be reported in the quarterly journal *Knowledge Creation, Diffusion, Utilization* [*7*], a publication for knowledge utilization and planned change. Thus, some professional interests are beginning to coalesce into socially responsible movements formed to promote the democratization of knowledge in both its sociological, as well as its psychological dimensions [*8*].

In helping to articulate this socioentrepreneurial movement, following Hall's model [*9*] of dissemination, some entity could be created to serve as: (*a*) a catalyst for melding theory and practice, (*b*) producing knowledge utilization, and (*c*) exploiting knowledge in the process of bringing about planned change. These outputs may be embedded in each of the following themes [*10*]:

- Creating a national learning community for improving human services.
- Knowledge utilization strategies in managing the changing human services milieu.
- Planning survival strategies for human services, programs, and agencies.
- Traning human services administrators for knowledge utilization and planned change.
- Defining public and private roles in bringing about planned change.

The multifaceted advances being made by researchers, policy makers, administrators, educators, consultants, and other service providers are being applied to knowledge production and dissemination. But little attention is focused on the kind of responsible knowledge utilization that seeks to reduce human loneliness and the anguish of being uninformed, as well as foster individual personal development. Indeed, the dichotomy between the celebrated benefits of the information society [*1*] and the actual realization of these expectations is approaching a mental dislocation among the masses of people in the population.

Self-informative behavior has long been considered to be at the base of the essential transfer competency in society and constitutes that set of skills that underlie all areas of ability development. Before the beginning of the modern world, the otherwise educated person was either a churchman or a politician. In the Renaissance, such a person may have been called a scholar printer or a universal thinker. During the subsequent age of specialization, responsibility for this person's development was either transferred to the institutions or went underground.

Since 1928 [*11*], when it first became evident that adults could continue learning throughout the life span, light has increasingly been shed on the fact that there are few, if any, adults who do not process information constantly in everyday life [*12*]. In addition, more and more scholars have become independent of institutionalized specialization [*13*] but, on their own, have developed a measure of specific and generalized ability to carry out fundamental IP. In this regard, the independent scholar has profited from self-referral to the various expert systems, whether manual or machine based [*14*].

The composure of people today can no longer be unprotected from an overexposure to the cult of information [*15*]. So many people are frantic over the appearance of events passing them by that the national character is being infected with an excess of impatience [*16*]. Reacting to "blipped" spots of information, they scurry around in a patch of newsletters for elusive "tidbits" that will give them an edge on others. But the titillation is like a runaway infection, a disease that leaps from individual to individual as they try to outdo each other in cornering that one "newsletter" on all other newsletters.

PSYCHOLOGICAL SYSTEMS

During the 1950–1960 era, an IP approach to psychology was being developed as one "window of opportunity" for explaining human behavior [17]. This focus on IP led many scholars to reexamine the older schools of psychology from a systems perspective. Instead of the hoped-for grand synthesis of learning psychology, two streams have developed in parallel fashion and, at times, have overlapped with one another: cognitive psychology and social learning theory [18, 19].

In the language of information analysis, these parallel developments are analogous to the two approaches of top-down and bottom-up analyses [20]. In the first, cognitive psychology, knowledge is considered in relation to a screen of human attributes (K_A). In the second, social learning theory, human attributes are considered within a screen of knowledge aspects (A_K). From the perspective of the actual human being involved in the affairs of a personal life style, circumstance may, on occasion, dictate both top-down (K_A) and bottom-up (A_K) approaches.

No humanly significant approach to IP can be taken by either K_A or A_K to the exclusion of the other. Humans may at one time face circumstances that force them into a top-down analysis of a problem. In other circumstances, they may be heavily involved in a bottom-up approach to situation resolution. Indeed, one may question whether there are two psychologies at variance with one another or a systematic continuance of IP behavior encompassing various psychologies within which any particular individual may range [12, 21].

In fact, from the syntheses of research about cognitive style, learning style, and the transfer of skills acquisition [22], it appears that individuals map themselves onto appropriate behavior sets, depending upon the circumstances surrounding their IP. Over the years since mid-century, when Witkin [23] first conducted experimental studies in perception, at least 19 separate polar dimensions of IP have been identified [24]. Thus, the complexity of human IP is evident, and the range of possible combinations is enormous [25].

Facilitating providers may have to enter the behavior manifestations of any individual, whether bottom-up or top-down, and move to any level therein as quickly and nimbly as the information user does [26, 27]. Citizen expectations, which seem to be rising exponentially, come at a time when the information professionals are slowly moving from resource use studies to investigations directed at IP in the human mind [28]. In addition, the experts in artifical intelligence (AI) have not been able to deliver the kind of proactive services and products based on natural language processing (NLP) that their press agents seem to have so glibly promised.

From an analysis of some psychological positions, one could get the impression that the processes of information utilization were largely subconscious phenomena in the makeup of any individual. More than likely, some are and require the facilitation of an assisting information consultant for articulation and development. But for the most part IP is a highly deliberative affair, with patterns guided by the imperatives emerging from the organizing circumstances in everday life [29]. Much of the current emphasis on self-directedness and individual responsibility has developed from the empirical foundation laid down by Tough [30] and from the top-down advances of Knowles [31].

In these instances, the focus has been on adult populations, where there has been a sharp increase in the number of self-directed learning (SDL) projects. The rise in IP for "decisioning," learning, and communicating has been fairly uniform across the population as a whole. This emphasis on the adult as learner takes on a considerable social urgency when the recommendations of the Commission on Higher Education and Adult Learners [10] are taken into consideration:

Developing or renewing employability for the unemployed.
Maintaining and enhancing occupational skills in the face of technological change.
Eliminating adult illiteracy (both literate and computer based).
Providing equal access to education for all adults (whether self-directed or teacher based).
Developing knowledgeable citizens in an information technology society.

These high hopes for human development depend in great measure on an informatively relevant response system that rests on a "volkswagen" approach to a personal computer and NL software. Work in AI is helping to advance the art, but the ventures taken, whether top-down or bottom-up, have yet to fully explicate the interconnections with real life interface. Empirical studies are called for that verify and validate a naive psychology and a naive physics and develop a frame of reference out of the actual way people are involved in everyday life [32, 33].

In other words, an explication is needed of the questions surrounding how a self-directed information processor uses everyday language to think about and express decisioning, learning, and communicating behaviors, both verbal and nonverbal, whether alone (intrapersonal) or with others (interpersonal) [34]. Variation in personal IP is probably as wide as there are individual humans with unique situational constraints and opportunities. However, the transcript analysis of verbal self-reports shows several patterns that are recognizable to most, if not all, human beings. Pattern recognition and use is a function of their membership in the human race, a particular culture, ethnicity, or country and its language.

Social learning theory [35], together with Lewin's [36] formulation of the concept of life space, presents the psychological, social, and physical elements as forces in a dynamic framework. Social learning theory would appear to be the consensus theoretical framework within which much of learning research (especially that on humans) will evolve in the next decade. Study of the life space focuses on action, change, or locomotion and works to join neobehaviorism with social learning psychology, thus adding learning by modeling to the learning by doing of self-directed learning [37].

SD-IP occurs when a person encounters some task or problem and the environment provides information concerning the nature of the problem or task, necessary knowledge, or skills for its solution and relevant performance criteria [38]. A moment of SD-IP thought such as this might occur during a brief lull in a bewildering array of interruptions, or the individual may not know what SD-IP task is needed far enough ahead of time. Information stored and indexed for one purpose today may be useful in a completely different way at another time. Technically speaking, the SD-IP user needs flexible databases with front-end NLP systems that allow for different record length, associations between records, and the content of records.

The development of such NLP software systems integrated for SDL purposes will be facilitated to the extent the SDL researchers identify the functions that are to be accomodated and communicate them to the knowledge engineers in developing expert systems. Of all the unique combinations of behaviors manifested in SDL situations, research findings continue to support a definable group of transferable skills that are significant across various situations [22, 39, 40].

Software "understanding" cannot be developed unless it is embedded with extensive knowledge of the particular world with which it must deal. It is relatively easy to develop mechanistic approaches based on tight logical systems, but these are inadequate when extended to real world tasks. The everyday world of IP is often messy and illogical; therefore, experts in AI have had to leave specialized subject approaches behind and become much more psychological [41–43]. But the problem remains of not having a real world contextual psychology within which cognitive psychology can be embedded.

Any prospective SDL software must be able to operate swiftly and efficiently in this human environment of organizing circumstances [29] if a microcomputer system is to be a personal SDL tool based on front-end NLP. In the practical world of SDL, the individual must be able to store, maintain and process all relevant information in integrated data bases. The wave of the futute is to have a personal data base system upon which "all" functions can be performed. Such a system would have to be able to manipulate both structured and unstructured data while, at the same time, capturing information at the time of thought.

Unfortunately, people are often dismayed that the possibility of the computer taking over cognitive processes is very real because of the great deal of research and data syntheses that have been done. On the other hand, the knowledge about perception and attending (sustained awareness) is more diffuse, susceptible to personal variation (e.g., cognitive and learning sytles), and patterned in systems of facing the world [44]. It is assumed, and some evidence exists for such a conculusion, that these patterns and styles are reflected in NL, which also can be embodied in software programming.

Few people seem to realize, however, how exceedingly complex and often evanescent these patterns of perception and attending really are. They have been the substance of the literature of civilizations and feed the diversity of human cultures. It is rare to find either a cross cultural approach to IP or an examination of the cultural variations in cognitive behavior. Computer software systems seem to have been more oriented to machine architecture than to cultural variation. One way of doing this is through the controlled analyses of guided introspective accounts of SD-IP.

The presence and thrust of cognitive psychology is a case of the mind studying the mind. It presumes to replace all of the previous psychologies by studying the foundation on which all other social sciences exist [45, p. 6]:

> Understanding how humans think is important to understanding why certain thought malfunctions occur (clinical psychology), how people behave with other individuals or in groups (social psychology), how persuasion works (political science), how economic decisions are made (economics), why certain ways of organizing groups are more effective and stable than others (sociology), or why natural languages have certain constraints (linguistics). Cognitive psychology studies the foundation on which all other social sciences stand.

Although cognitive psychology is empirical, it eschews the limitations of behaviorism and aims to replace the humanists and the gestaltists. Its principal methodology, protocol analysis, and introspection stems from the work of Wilhelm Wundt in the last quarter of the last century [45, p. 7]:

> In this method, highly trained observers reported the contents of their consciousness under carefully controlled conditions. The basic belief was that the workings of the mind should be open to self-observation. Thus, to develop a theory of cognition, one needed only to develop a theory that accounted for the contents of introspective reports.

The use of such a methodology has raised a number of confusing issues such an imageless thought, the duality of perception and imagery, and the question of whether memory is dual, multiple, or neither. The confusion has compounded the problem of the representation of knowledge in memory but especially in external devices such as software analogs. In addition, to compound the matter, there appears to be evidence for an abstract, nonsensory code, as well, as how this abstract information might be represented [46].

The rise and prevalence of behaviorism has been taken as evidence of the limitations of introspection: its irrelevance and its apparent contradictions. Fundamental to the problems was the lack of theories explanatory of internal cognitive processes. After mid-century, various influences were tapped for heuristic models:

- IP advances that grew out of information theory and human factors research on human skills and performance.
- Computer science efforts to get computers to behave in a manner resembling human intelligence. Numerous computer science concepts have been indirectly applied to psychological behavior.
- Linguistic structural analyses showed that language behavior was based on higher mental processes more complex than could be accounted for by the prevailing behavioristic formulations.
- Neural network simulation research, employing software designed around the random association of infants learning NL behavior.

Work in these fields has led to improvements in methodology over introspection as the only source of empirical evidence. Measures today are commonly taken on frequency of success in a task and on performance speed, which are aimed at improving human cognitive behavior. Task success is usually expressed in percentage correctness to make comparisons less arbitraty, and speed is referred to as reaction time. Statistical significance is coupled by a reliability measure (i.e., reproducibility), and the importance of the difference is determined on criteria applied to both reaction times and performance differences.

There appears to be a tendency to employ AI software as the source of data for measures of performance and reaction time. Obviously, cognitive psychology is directed toward those theories that hold explanatory power for cognitive skills and behaviors. The major model of human cognition appears to emphasize IP defined within such systems boundaries as:

Representation of knowledge, neutral imagery, stored information, schemas, and prototypes.
Memory and learning, elaboration and reconstruction, cognitive skill, and declarative and procedural knowledge.
Problem solving and reasoning, induction and deduction, language comprehension, and generation.

This emergence of a shift toward abilities other than cognitive ones draws new attention to the complexity and pervasiveness of both perception and attention. Arguments associated with these matters advance the notion of multiple simultaneous thought processes within which the serial problem solving is a special case [47]. Whether obtained from introspection or laboratory experiments, IP becomes more persuasive if one steps back from the cognitive domain and observes the broader perspectives of human behavior [48]. In fact, thinking as executive cognition may not only be parallel but so distributed in processing as to exist only as a theoretical model and explanatory only of the currently popular computer software approach to intelligence [49].

The computer metaphor of cognition may continue as a mainline approach to research methodology, but the re-emerengence of verbal reports and introspection can help to enrich the totality of research findings [50]. Protocol analysis of observations has proved valid for the recall of the contents of human behavior in incubative and intrusive thought, (e.g., contents of focal attention, current sensations, plans,). But to bring introspection to these contents would be to deny the ability of people to be involved with metacognition [51], that is, the thinking about the processes of thinking [52].

Focusing on computer IP as the model of cognitive problem solving has tended to block an understanding of the way in which perception and attention may have contributed to successful outcomes. Performance has traditionally been evaluated on such factors as the time taken to solve a task and on the correctness and elegance of a solution. But such approaches have obvious limitations when the aim is to nurture the underlying life style performance of an individual or in order to promote the transfer of problem-solving ability [53].

A critical attribute of a theory of perceptual independence is that it have a separate structure devoted to both perceptual and decisional processes. Perceptual independence is, by definition, a perceptual phenomenon and is defined as an attribute of the perceptual system. Extra assumptions are needed (e.g., whether processing is serial or parallel, self-terminating or exhaustive), and these make the theory more controversial [54]. A fundamentally important problem is to determine how these dimensions are combined in perceptual processing [55].

Unfortunately, perceptions are not usually directly observable; instead, they first pass through some decision process that uses the perceptions to select a response appropriate to the general experimental milcu. Decision or judgment processes, therefore, fundamentally alter direct perceptions (i.e., the early stages of perception). The theory of perceptual behavior is a substantial generalization of signal detection theory, which can account for experiments with stimuli composed of two or more components and with any of a wide variety of response instructions [56].

A decision problem is defined by the acts or options among which one must choose, the possible outcomes or consequences of these acts, and the contingencies or conditional probabilities that relate outcomes to acts. The term "decision frame" is used to refer to the decision maker's conception of the acts, outcomes, and contingencies associated with a particular choice. The frame that a decision maker adopts is controlled partly by the formulation of the problem and partly by the norms, habits, and personal characteristics of the decision maker [57].

It is often possible to frame a given decision problem in more than one way [58]. Alternative frames for a decision problem may be compared to alternative perspectives on a visual scene. Rational choice requires that the preference between options does not reverse itself with changes of frame. Because of imperfections in human perception and decisioning, however, changes of perspective often reverse the relative apparent size of objects and the relative desirability of options.

When faced with a choice, the "rational" decision maker will prefer the prospect (theory) that offers the highest expected utility. A predictive approach encourages the decision maker to focus on future experience and to ask, "What will I feel then?" rather than "What do I want now?" The former question, when answered with care, can be the more useful guide in difficult decisions. In particular, predictive considerations may be applied to select the decision frame that best represents the hedonic experience of outcome [59]. In addition to consistent criteria, the common conception of rationality also requires that preferences or utilities for particular outcomes should be predictive of the experiences of satisfaction or displeasure associated with their occurrence. Thus, a man could be judged irrational either because his desires and aversions do not reflect his pleasures and pains. The predivtive criterion of rationality can be applied to resolve inconsistent preferences and to improve the quality of decisions [60].

Cognitive styles represent an integrated component of an individual's mode of functioning and are evident in the ways that individuals respond to situations and circumstances. Cognitive styles are broad stylistic characteristics that individuals employ to process information within their cognitive structures. An individual's cognitive style determines the method used to apprehend, store, and use information and refers to the individual's different approaches to understanding, remembering, and thinking [61].

Research is beginning to reveal extreme cognitive style polarity in the skill acquisition of many information providers [62]. Question negotiation involving direct interpersonal interaction seems to be characteristic of field-dependent practitioners. On the other hand, field-independent providers appear to excel in search strategy retrieval, particularly for the more complex knowledge inquiries. Such bicognitive differences in staff behavior also include those listed in Table 1.

At least 19 cognitive styles have been identified [24] that describe the various dimensions within which individuals structure different perceptual and problem-solving functions. Various assessment techniques have been developed; and although these numerous techniques are correlated with one another, the relationships are far from perfect. Different dimensions of cognitive style, which seem to be measured by each, may be a function of the item variance that occurs in test proliferation rather than in the basic measures of rod-and-frame and embedded figures tests [23, 63].

TABLE 1, Differences in Cognitive Style

Variables	Field = Dependent Individual	Field = Independent Individual
Peer Relations	Seeks work with others for common goals.	Task oriented; not attentive to social groups.
	Likes to assist others.	Prefers individual work.
	Sensitive to feelings and others opinions.	Likes to compete for individual recognition.
Personal relations	Expresses positive feelings openly.	Restricts interactions to tasks at hand.
	Needs mentor guidance.	Eschews physical contact.
Supervisory relations	Seeks guidance and personal contact.	Prefers new tasks without the help of others.
	Seeks supervisor rewards.	Seeks nonsocial rewards.
	Only motivated by one-on-one contact.	Impatient to begin and finish tasks alone.
Information seeking	Seeks global knowledge.	Seeks concept details.
	Humanized or fictional; concept presentations.	Cognitive (math, science) presentations.
	Proactive deliverables.	Reactive deliverables.

In his research, Witkin [*23*] noted that certain individuals relied heavily on the outside environment for perceptual cues, even as these conflicted with internal ones. Others were able to separate easily essential information from a surrounding visual field. The two orientations, labeled field dependence and field independence, respectively, exist on a continuum, with individuals found at all points. Field-dependent individuals tend not to add structure to visuals and accept visuals as presented. Because they do tend to fuse all segments of a visual field (e.g., a picture), they do not view the visual's components discretely.

Field-dependent and independent individuals approach learning in different ways [*64*]. First, field-independent individuals, being more analytic in approach, tend to act upon a stimulus complex, analyzing it when it is organized and structuring it when it lacks organization. In many instructional situations, metacognition, or the ability to analyze and structure aids in learning, is lacking. The field-dependent learner, however, takes a more passive approach, accepting the field as given, and experiencing it in a more global, diffuse manner.

This passive approach means that field-dependent individuals tend to notice those cues in a stimulus field that stand out or are more salient. When the stimulus is arranged so that the salient cues are also relevant, the field-dependent person may experience little difficulty. In fact, if a

learning task is clear, well structured, and low in complexity, there may be no significant differences in learning by the two orientations. In situations where cue relevance and saliency are in conflict, the performance of field-dependent individuals seems to suffer.

A visual with a background containing certain relevant (contextual) cues would be of value to field-dependent individuals in recalling information about figures appearing in the picture or visual. Compared with field-independent persons, field-dependent individuals have a greater need for, and are more dependent on, external sources of structure (background) and organization. Information recall from visuals for field-dependent individuals is facilitated if major visual cues are made relevant. It is hindered if important appearing cues are irrelevant or not noticeable.

The field-independent person samples more fully from cues, both salient and nonsalient, and performs more successfully. Increasing salience by manipulating instructional material should tend to make the field-dependent learner sample more fully from all cues, thus modifying the visual strategy employed. Field-independent persons tend to be able to receive information from both relevant and irrelevant cues. If field-dependent individuals are shown a procedure (modeling) in which they view figures within a background containing important and useful cues in terms of number, shape, and location, they will be able to transfer the context to these types of pictures on subsequent viewing of pictures or visuals without these relevant contextual cues.

Because of the ability of field-independent persons to create structure and not be dependent upon external cues, they should not be affected by either treatment (presentation or order). Field-independent individuals should score higher than field-dependent individuals on any visual recall test. If information can be identified that supports the concept for certain types of cognitive styles, certain visual formats become more effective. Field-independent individuals tend to give structure to unstructured visual material and tend to separate an individual item or component from its overall context. Field-dependent individuals, on the other hand, tend to respond holistically to stimuli.

The complexity of cognitive style derives from the many dimensions (19) identified by Messick [*24*] out of the many dimensions that constitute the polarities of field dependence/independence. Thus, the characteristics of cognitive style cannot be limited to a single dichotomous continuum but constitute an environment of several polar dimensions. However, for initial discussion purposes, it is helpful to view behavior from the dichotomy of the seminal and macropolar dimension, as measured by the group-embedded figure test (GEFT).

SD-IP THEORY AND RESEARCH

Much has been written about SDL research and theory generation and it is instructive concerning a phenomenon that has captured the interest and involvement of the general population [*65*]. Of the numerous reviews and syntheses of SDL the most useful are those that place it in a continuum of lifelong human development [*66–68*]. The life wide impact of SDL stems from the various networks within which individuals process information in the community [*69*].

Much of what the adult does in SDL is in reaction to, or as a result of, sequential conditioning, life-style, growth, and successful achievement. This "learning stance" is comprised of a number of dimensions [70].

Learner choice, responsibility, and ownership: the learning agency.
Enabling settings and enablers: a community of self-directed learners.
Participating conditions: IP for decisioning, learning, and communicating.
Ownership of learning processes and products: proprietary versus shared values in community.

Personal existence encompassed within a life style includes all the facts within the life space at any given time that have existence *within the perception of the individual* and excludes those that do not [36]. These elements not only constitute the field or life space of the individual but also the boundary zones. Other facts, events, or people in the social and physical world beyond those boundaries do not affect the individual involved in real life circumstances at that time [29].

The changed circumstance tends to provide a single or at best, very few resources or opportunities for learning that are reasonable or attractive for the learner to pursue.
The structure, methods, resources, and conditions for intrapersonal IP are provided or dictated most frequently by the circumstances.
Learning sequences progress not necessarily in linear fashion but rather, as the circumstances created during one episode become the circumstances for the next necessary and logical step in the process.

Individuals bring to each episode or project their own motivation, aptitude, creativity, energy, and tenacity. The essential elements for understanding the process appeared to be (*a*) the expectations of the individual, (*b*) that person's inventory of skills and knowledge, and (*c*) the particular resources present within the environment. Contrary to the evidence of numerous correlation studies, the demographic characteristics of the learners appear to be less important than the uniqueness of the individual's circumstances [21].

The following is less of a model of learning, with precise steps or directions for a particular investigation, than it is a guide that has proved useful for the study of SD-IP projects.

The study of SD-IP projects profits from a naturalistic or qualitative approach based on interviews with individuals. Such projects are inherently individualistic and can be understood primarily in terms significant to the individual.
A detailed account of the project is required and is most likely to be described in the chronological order in which events took place even though they are not functionally related. Probing questions are almost always necessary. At the end of the interview, the individual should be asked which events had influence in reaching a final goal.
Individual clusters of elements tend to emerge through the labeling process because clusters either reach a dead end, produce a singular product, or lead directly to the origin of another cluster.

With a transcript of the interview in hand, clusters may be identified as separate entities or as parts of a series of related clusters.
Once identified, each individual cluster can be analyzed and understood according to how the elements interact and affect one another.

These analyses give insight into the degree of significance that respective elements contribute to any SD-IP project. For instance, residual knowledge has been found to have a part in project development. The analyses may demonstrate that this prior knowledge was a major factor in only a few instances for determining the final product. However, cluster analysis has identified several cases in which fortuitous environmental factors occurred, and in a majority of these instances they became the base for launching the SD-IP project. At the same time, the amount of exploratory action initiated by the individuals may vary from a great deal to very little with no readily discernible explanation.

The data obtained through interview and protocol analysis may be better understood in application to selected activities drawn from SD-IP project descriptions. Any real life SD-IP project is composed of clusters of elements, and it is necessary to study these clusters individually, as well as in relation to the project as a whole. A cluster is a defined set of elements within the organizing circumstances of real life that have interactive relationships. These elements include (*a*) knowledge (residual or acquired) (*b*) actions (directed, exploratory, or fortuitous) and, (*c*) environment (consistent or fortuitous) [71].

One may dismiss the significance of these processes in real life as serendipitous. But the sustained attention of a human being as information processor, based on values and processor, based on values and purposes, may be considered as analogous to the preplanning of an instructional developer. However, rather than preplanning their projects as teachers and librarians, SD information seekers and processors tend to select a course of action from limited alternatives that occur within the environment. Their behavior is guided by interpersonal discourse and the nonverbal (sight, sound, movement) manifestations of sociodrama and leads to significant inferences.

Thus, the behavior of the individual, as identified and investigated by Tough [30, 72] and others, reveals such intangibles as information processing, imagery, problem solving, vicarious modeling, and motivation. In a learning-by-modeling process, Bandura [35] identified the behaviors of observation, attention, retention, motor reproduction, and motivation in learning by modeling. However, Bandura did not give attention to the structuring of what was (or is) processed. As a result, in a series of studies (1982, 1984, 1987), Spear and Mocker have drawn out the implications for SD-IP.

SD-IP project descriptions that have been protocol analyzed to identify the suggested elements of environment, behavior, and personal attributes and to search for other factors suggest the following organizing circumstances [29]:

Anticipated single event that the individual enters, expecting that IP and possibly learning will be required, but has little or no idea of what must be processed or, specifically, how it can be learned.

Unaticipated single event where tasks are performed by people repeatedly but where the individual does not anticipate being engaged in a learning project.
Series of related events that are not anticipated but where each episode in a fixed sequence provides the organizing circumstance for the one that follows.
Series of unrelated events in which individuals assemble random bits of information, observations, or perceptions whose purpose for retention emerges over time.

The impetus or triggering event for IP or episodic learning stems from changes in life circumstances. Such change may be positive or negative, may happen to the individual or to someone else who impacts that person's life or may be an event that simply occurs and is observed within the life space of the individual. Following Lewin [36], life space is defined as the physical, social, and psychological environment in which the individual lives and functions.

In application, a SD-IP project, whether based on a "shopping list" or not, is more the process of searching for and acquiring the materials than the building of "the house." Materials are seldom acquired in the order of their use, and their relationship may not be apparent until the project, or house, is finally built. The activities in a SD-IP project are clusters of elements that usually result in acquiring resources that are stored, at least for a time, until they fit into, or are used in relation to, other acquired resources. Each cluster of elements has its own determinants and effects and must be first studied as a discrete entity.

The social dimensions of these impediments should be a part of the sociology of the information professional and of learning in general, and serve as matrices for the questions of policy development [73]. The implications of SDL are as revolutionary as the results of the information and communciations revolutions [16] that have ushered in the TELCOM (telecommunication) society. Long [74] goes on to assemble evidence that cognitive development continues across the life span. Both Kirby [22] and Knapp [75] document the support for the life wide occurrences of transferrable skills.

Various syntheses of empirical findings derived from SDL research exist [76–80]. Although these deepen and extend the pioneering work of Tough [30, 72], the problem of SDL "instructional" applications remains as underdeveloped, as Hilgard and Bower [17, Chapter 16] have pointed out for other-directed educational systems. Brookfield [81] assembled a group of scholars to address this issue of a disparity between SDL theory and its relation to field practice.

Social learning theory, as developed by Bandura [35] and others, is assumed [17, pp. 599–605] to provide the best integration of modern learning theory to the solution of practical problems. Hilgard and Bower [17, pp. 21–23] propose a series of questions that, if answered, could serve to help integrate learning theory into a cohesive system. Others [82, p. 5] maintain that the theory has overextended its empirical base:

> It would appear that we have done serious work in conceptualizing the nature of the method—especially as evidenced in the work of Tough [30, 72], Knowles [27, 31] and Spear and Mocker [29]—but have not gone far enough in our empirical work.

The models of self-directed learning (as individually determined) can be functionally related to, if not built on, a foundation of theories of IP. This is seldom done, almost as if self-directed learning had such special problems as would make it unique. Other researchers have noted this lack of consideration in the literature; for example, Mocker and Spear [83] have developed a four-part differentiation based on their investigations of organizing circumstances within which various types of learning are embedded.

These organizing circumstances are determined by, and embedded in, the constraints of the environment. Organizational circumstances seem to account for as much decisioning, learning, and communicating competencies as to do the personality dimensions of cognitive style [23]. These dimensions yield procedural patterns that are closely aligned with steps that have to be taken to offset the "missing link" [84], faced by all too many adults in real life and for the elimination of which, supposedly, the professions had a social responsibility [10].

The verbal report of each respondent's efforts to resolve a situation or solve a problem should be carefully collected and analyzed for patterns [50, 51]. In examining the transcript or in critically listening to the audio record, the researcher gains awareness of strategies selected, thought patterns employed, memory capacity, and solution monitoring ability. The parameters of the exploration are determined by the interview schedule or instrument used, the task demanded of the respondent, and the processes to be explored and categorized.

Behavioral areas can be explored individually or in combination, as each respondent struggles to resolve a situation or achieve some task requirements. There is scarcely any limit to the type of question or tasks that can be explored. However, only the concomitant use of experimental and ex post facto research designs will offset the major criticism that clings to the method since Wundt [85] first used it:

Behaviorist skeptism about the use of retrospection, introspection, and protocol analyses unless coupled with more objective measurement.
Wide methodological variations have resulted from loosely applied criteria that fail to differentiate between legitimate and illegitimate techniques.
Tape-recorded transcripts yield hard "objective" data of the respondent's original statements, which can be analyzed by a number of researchers to achieve inter-rater reliability.
Cognitive processing models have strengthened the theoretical structure of the analyses and the logical development of interpretative hypotheses.
Preprocessed data segments removed from extraneous elements can be encoded into the language of the theoretical model for category selection and assignment.

Present trends in SDL research leave out many areas that have been claimed to demand attention [81] and that could be of advantage to information professionals in developing a client-centered psychology. Librarian research seems to be influenced, sometimes heavily, by fashion, which may help to explain the neglect of such important and significant studies as Peters [51] and Adult Basic Education (ABE) life style research [86]. In fact, Peters [51, p. 2] specifically drew attention to this relationship:

We are especially interested in the dynamics of learning in "natural" or nonformal settings and in the role that literacy plays in learning. All of this is couched in a problem-solving framework, consistent with our belief that learning is the result of a problem-solving process.

Integrating SDL research findings into cognitive pychology conceptualizations and IP theory across the life span is directed at answering and addressing the issues raised by Huey Long and his colleagues at the University of Georgia [*74*,pp. 11,12]:

> Educators of adults generally appear to be of the opinion that cognitive development continues into and across adulthood. Individuals interested in the education of adults, however, continue to search for a cognitive theory that addresses life span development.

This issue must be addressed squarely, and a stand has to be taken based on the evidence of research. Such a position could accept the following assumptions to advantage. The transferrable competencies, described by various reviewers, do not occur by happenstance but are deliberately applied to meet SDL objectives [*87*,p. 15]:

> Learning is a problem-solving process in which a new idea comes into the perception of an adult. The adult interacts with that idea mentally trying or deciding not to try to give the new idea a chance at entry into the memory. The adult makes a conscious decision to accept or reject each perceptual input. Learning is not ever all that new because there must be some way to relate new ideas to prior ideas. A whole new idea still needs to get a foothold in the mind based upon an old or prior idea.

Few information professionals would question the validity of this process when directed by a teacher where external observation and verification are supposedly applied. But those same critics who place confidence in the reliability of a teacher-controlled classroom behavior vociferously question the objectivity of the processes when applied by individual "street persons" in real life. In other words, the validity of the learning process is somehow abrogated when it is applied by adults in everyday SDL contexts that occur without benefit of teacher intervention.

Such a contention is not only challenged but refuted by the findings of SDL research. From Tough [*30*] through Reisser [*88*], Knowles [*31*], Ziegler [*70*] and beyond to Gibbons [*89*], Peters [*51*], Cross [*76*], Spear and Mocker [*29*], SDL processes exhibit individual propose, planning, and integration and are deliberately aimed at the objectives of th learner. With Peters' [*51*] study as with Tough's approach, the significance cannot be overestimated. Despite the fact that these investigations are almost totally ignored by librarians, it is likely that any future IP research will have to take them into account when the validity and relevance of other research findings are considered.

In retrieving and subdividing a set of data for an individual to use, it is necessary to attend to the psychological nature of that data. Humans prefer to specify and be told about subdivisions and relations that they already understand. Appropriate and meaningful access paths depend on the human user's knowledge structure, which is not necessarily the most convenient for systems design.

Structure is the relation between different elements of knowledge and the constraints on the ways people can think about them. At all times it is necessary that particular attention be given to bridging the gap between the world of the laboratory and our experience in everyday living. The only way to organize empirical results effectively is by theory; but all too often the advances taken are aimed at college students and those with further education.

Almost nowhere can an approach to cognitive psychology be found from the person-on-the-street orientation, granted that a study of cognitive psychology will help the student to improve the capacity of their intellects and maximize the consequences for intellectual performance [*90*]. If it is so important for people to improve the way they think by understanding the basic mechanisms of human thought, why not create the software to help those who do not have the time for such discovery excercises?

PROFESSIONAL IMPERATIVES

Suffused with exaggerated claims of AI and the software control of human beings, people need to know the limits of the computer takeover and the opportunities offered by machine-assisted human development. The computer analog of software IP may account for many automatic and deliberative cognitive processes, but these behaviors are but the workhorse of a rich and varidimensional life-style permeated with many awareness and attentional manifestations. There are so many of these that one can only identify their massive dimensions appropriately as cultural and intercultural environments.

How will the gap be bridged and the missing link be filled between the people's concerns and the information entrepreneurs [*91*]? Obviously any breakthrough will not be a single dimensional advance but a combination effort of many interests, including the knowledge industries, information technocrats, and the information welfare systems as well as the associated professional organizations. In addition, although no single information service or product will predominate, research and development centered around a "volkswagen" standardization of people's access is both a marketing and socially responsible necessity.

Research and development, of course, continues to be advanced on various fronts by any number of vested interests. Ostensibly, these efforts take the consumers' needs for information services and products into account. Unfortunately, however, proprietary matters all too often crowd out that research and development upon which the standardization and simplification required by the masses of people in the population can be based [*92*].

Such hopes, it appears, have been the marketing promise of the information industry—that it can make computers see, talk, listen, and think like humans in everyday life [*1*]. Although the AI community may scoff at these statements, professionals have done little to take into account the real life information processing of the vast numbers of people in general population. The cognitive model of computer software processing represents only a part of people's life-style information behavior.

Every person constantly produces knowledge; it is a function of continual IP, whether applied to decisioning, learning, or communicating. The output behavior of any individual is a state of equilibrium among the processes that satisfies the decisioning, learning, or communicating constraints.

Thus, IP functions to liquefy knowledge (whether retrived or recalled) in order that it be reconstituted or "solidified" around a new equilibrium.

Each equibibrium represents a resource (whether stored in concepts or in manual and machine-readable form) that can be referenced when future knowledge "liquefactions" are needed. Any interactive and intelligent front-end access system would seem to have to be built on principles at variance with those that stem from rigorously defined cognitive problem solving. The research in self-directed information processing appears to hold promise in extending application to these developments.

In traditional information and library science, the taxonomy of the cognitive domain [93] has been fairly widely employed for indexing schema and data base development. Today, the affective and psychomotor domains, although behaviorally taxonomized [94, 95], are being brought into professional consideration through the "back door," so to speak, by the recent emergence of cognitive psychology. Cognitive psychology could be described as a system approach to rational problem solving—combining aspects of various older psychologies such as behavioral, humanistic, and gestalt. The prevailing metaphor in cognitive psychology for studies of learning and memory emphasizes the acquisition, storage, and retrieval of information.

On the other hand, mind can more realistically be described in terms of skill in manipulating processes; the notion of skills provides a useful framework for accounting for the significant aspects of intelligent and interactive information. Evidence supporting the procedural view includes studies that show that (*a*) the means for the acquisition of information form part of its representation in mind, (*b*) recognition varies with the similarity of procedures in acquisition and test, and (*c*) transfer between tasks varies with the degree of correspondence of underlying procedures.

To factor one's total approach to an IP psychology into encoding procedures, retrieval processes and their interaction based on a computer metaphor of problem solving [54] are a somewhat limited approach to an intelligence that is artificial [15]. The fact that space and structure are a large part of both the traditional library's and the computer's operations is not a sufficient reason to suppose that such a dimension alone describes human thinking adequately [96]. Observations such as these should not be construed as a Luddite protest but should only serve as a reminder that "A long tradition has held that 'contents' of the mind in some sense exist independently of the 'processes' that create the contents, change them, and make use of them" [97,p. 265].

Focusing on computer IP as the model for cognitive problem solving has tended to block an understanding of the way in which the culture of perception and attention may have contributed to successful outcomes. Performance has traditionally been evaluated on such factors as the time taken to solve a task or the correctness and elegance of a solution. But such approaches have obvious limitations when the aim is to both nurture the underlying life-style performance of an individual and promote the transfer of problem-solving ability [53].

On the other hand, the knowledge about perception and attending (sustained awareness) is less well documented, more susceptible to personal variation (e.g., cognitive and learning styles), and patterned in systems of situation-facing involvement. Few professionals seem to realize how exceedingly complex and often evanescent these patterns of perception and attending really are. Such deep abiding impressions have been the substance of the literatures of the civilizations and continue to feed the diversity of human cultures.

It may be an oversimplification to say that once perception and attention are accounted for (controlled by some input coding device), cognitive psychology explains the bottom line of those cognitive processes that operate in all human beings. However, cognitive processes appear to be a common core of intellectual support behavior that can be programmed in software embodying declarative and procedure knowledge operations. This common set of cognitive skills that need to be learned by all persons retrieving informative data, regardless of perceptual style and attending patterns, can now be delegated to machine learning.

The intellectual structures built upon such foundations as cognitive psychlogy are impressive, as well as the AI applications in computer programs and software systems. But suspicions about this magnificent superstructure become evident when one examines the actual empirical base of facts (variables) and relationship that have largely been obtained from observations limited to white mice, Pavlovian dogs, and college students. Almost nowhere has a fresh look been taken at the behavior of adults involved in the curriculum of everday life.

Although no one common definition of intelligence exists, contemporary research in the field concentrates primarily on the metaphor of IP, even though cultural contexts and their interrelationships [52] are presumed to be included. On the other hand, some considerable progress has been made in conceptualizing intelligence, at least in the cognitive domain and to the point where aspects of it can be programmed in computer software [38]. On the other hand, an interest has emerged concerning real life intelligence, particularly among adults immersed as they are in life span life-styles and in the affairs of everyday life [12, 30].

This emergence of a shift toward abilities other than cognitive ones draws new attention to the complexity and prevasiveness of both perception and attention. Arguments associated with these matters advance the notion of multiple stimultaneous thought processes within which serial problem solving is a special case [47]. Wheter obtained from introspection or laboratory experiments, the evidence for multilevel models of human IP becomes more persuasive if one steps back from the cognitive domain and observes the broader perspectives of human behavior. In fact, thinking as executive cognition may not only be parallel but so distributed in processing as to exist solely as a theoretical model and explanatory only of the currently popular computer software approach to intelligence.

The software analog of cognition may continue as the main framework for AI research methodology, but the re-emergence of introspection can help to enrich the totality of these and other research findings. Introspective observations have proved valid for the recall of the contents of human behavior in incubative and intrusive thought, for example, contents of focal attention, current sensations, and plans. But to limit one's research to introspection rather than the protocol analysis of these elements would be to deny the ability of people to be involved with metacognition [51], that is, the thinking about the processes of thinking [52].

Part of that theory can be derived from self-directed learning research [29, 30, 51, 98], and part is supported by research in cognitive psychology and problem solving [45, 53, 99, 100]. But the findings of investigations of precognitive behavior remain either dispersed in the literature or selectively codified by various schools of thought and for particular professional interests [101]. In addition, even the traditional information scientist feels more comfortable with the computer-coded output manifestation of

cognitive problem solving than with the input behavior of attention and systained awaremess.

In any event, from Binet [102], through Duncker [103], and Ericsson and Simon [50] to the most recent investigations of self-directed IP, "thinking aloud" protocol analysis obtains actual precognitive behavior data rather than subjective inferencing based on a priori specifications of software-coded problem solving. Within the professions themselves, the client–professional relationship could have a more humanly relevant explanatory power.

Of what value would such a revolution be in the information professional's psychological thinking? Today, despite industry claims to the contrary, the recent report of the Center for Libraries and Education Improvement [8] found that little attention was being given to the study of information use as distinct from library use. The realignment of priorities noted earlier by Cuadra [104] is an aspect of cognitive psychology that has to be developed before client-centered facilitations are generally available to the American public.

Information advocacy and resource validity would then become more generally accepted characteristics of intelligent consumer reports. Rather than the laissez-faire and largely amoral proliferation of deliverable characteristics of a bookstore, information clients could expect more responsible guidance and even wisdom applicable to knowledge transformation and utilization. With a more humanly significant psychology of life-style knowledge utilization, the unfortunate gap between the expectations aroused by the information revolution [1] and the reality of consumer information advocacy [4] could be narrowed more readily.

CONCLUDING PERSPECTIVE

Obviously, another revolution is in the making, perhaps of even more radical implications for social change than the previous information upheaval. If managed in a sociopsychologically significant manner, this marriage of knowledge and action holds enormous significance for the person on the streets of everyday life. But, unfortunately, the implications are still more sociological than psychological, supporting a top-down movement to be developed and administered by technocrats.

SD-IP has had a long history in human affairs. This capacity for self-development may be as old as the human species. Before the establishment of the institutions in society, it was the major form of education and the learning of how to process information most effectively for decisioning, learning, and communicating. As such, it could remain the principal accompaniment of the human animal struggling to emerge from the past.

In the mind of the self-directed learner, each piece of information relates to other lists of data in various ways. Each individual remembers and organizes data for a variety of purposes and functions, which can be identified and meaningfully organized in conjunction with appropriate software developers. Working together, this team can turn the SDL content and processes into the software representations and constraints required for NPL—front-end software packages that facilitate computer access and use.

That the ordinary citizen in everyday life has not been able to do so is but a result of the fact that NL front-end services are not generally available as yet. But that is only a matter of time and the further penetration of the population by personal computers. Even though the most revolutionary impact has been technological, the low-cost personal computer has made it possible for the average citizen to become the scholarly communicator whenever and wherever located. Like McLuhan's proto image of the irridescent light source, all information is available from any direction for processing and can be accessed at any other point in return.

The problem, of course, is the extraordinary complexity of the human being who, in a single person, may range daily across all of the known dimensions of IP in response to an opportunity or a confrontation. In the past, human behavior has been compartmentalized all too often in order for the information specialist and instructional psychologist to respond more easily. Self-directed individuals involved today in life-styles development expect to have their behavior tracked and responded to wherever they are in the behavioral cycle of decisioning, learning, or communicating.

The SD-IP practitioner working to develop the materials and methods of knowledge systems wants to know how precepts are structured in the human mind, how such concepts develop, and how they are used in understanding and behavior. The software expert works to program the computer (software development) in such a situation-producing manner that it can understand and interact with the outside world. The common problem in developing intelligent and interactive systems is to emulate the human conceptual mechanisms that deal with perception, image making, and language.

Within this framework, mind is often treated as if it were a physical object, and information, perhaps subconsciously, is assumed to have concrete properties. Mental processes are often described as objects or events in an actual physical space, as when we speak of storing and organizing memories, of searching through them, or of holding or grasping ideas in our minds. Like objects, memories may be lost or hard to find. All such descriptions are little more than fanciful nominalizations, (e.g., administrative organizatinal charts), clients), irrespective of the detail embodied in the accompanying propositions.

On the other hand, mind can be described more realistically in terms of skill in manipulating symbols. The notion of skills has been shown to provide a useful framework in accounting for significant aspects of cognitive processes. Evidence supporting the procedural view includes studies showing that (*a*) the means of acquisition of information form part of its representation in mind, (*b*) recognition varies with the similarity of procedures in acquisition and test, and (*c*) transfer between tasks varies with the degree of correspondence among underlying procedures.

Equally important, the everyday affairs of most SDL individuals are marked by frequent interruptions, both from outside sources and by internal prompts such as the pressure of a life-style task that has to be able to respond to an interruption and then return to the SDL behaviors at hand, as well as move easily from one task to another. Otherwise, many SDL neophytes, whether because of prior inhibiting experiences or lack of ability to follow through, may stop trying to perceive or hold interest in some new idea.

Information professionals could, if motivated, redirect toward broader populations the principal methodology of research in cognitive psychology—protocol analysis and possibly introspection—which stems from the work of Wilhelm Wundt in the last quarter of the last century [85]. In this method, randomly selected and trained observers could report the contents of their consciousness under carefully controlled experimental conditions. The basic belief is that the workings of the mind can be open to self-observation.

Thus, to develop a theory of cognition out of the interpersonal events of everyday life, one needs to develop a theory that accounts for the contents of introspective and protocol analysis reports.

The frequency of recommendations are increasing that the GEFT [23] be employed for observing field dependence/independence along with protocol analysis, rather than the numerous verbally based tests that have appeared in recent years. The GEFT provides a more valid and reliable measuring standard than the many microdimensional variants that may be used for in depth probes of those particular aspects of the 19 or more polar dimensions. The findings will thus enrich with examples the numerous characteristics of the complex and multidimensional construct of field dependence/independence.

Work in the field could lead to improvements in methodology over simple introspection as the only source of empirical evidence. Employing experimentation measures of success on task behavior and performance speeds would indicate real life information processing competencies. Task success, as expressed in percentage correctness and speed as reaction time, makes comparisons less arbitrary. Such participatory research in everyday human environments would offset the tendency to employ AI software as the only source of data about performance and reaction time.

Without methodological advances such as these or others, it is difficult to give much credence to self-directed learning as a theoretical construct. Of all the issues and topics discussed in SDL, there is one that occurs over and over again: SDL processes occur by happenstance and not under the carefully contrived administration of a teacher. This issue stems either from an ignorance of what happens during SDL processing in real life or from a lack of integration of research findings on the part of the pracitioners involved.

The computer metaphor of IP may account for many automatic and deliberative cognitive processes; however, these behaviors are but the workhorses of a rich and varidimensional life-style permeated with many awareness and attentional manifestations. Unfortunately, all too many people are dismayed with the possibility of the computer taking over cognitive processes, even though a great deal of research and data syntheses have been done. Suffused with seemingly exaggerated clamins of an AI and the software control of human beings, people need to know both the limits of the computer takeover, as well as the very real opportunities offered by machine-assisted human development.

REFERENCES

1. George Harrar, ed., "Celebrating the Computer Age.", *Computerworld, 20*(44): Special Section, November 3, 1986.
2. Fritz Machlup, *Production and Distribution of Knowledge in the United States*, University of Princeton Press, Princeton, NJ, 1962.
3. Fritz Machlup and Una Mansfield, ed., *Study of Information*, Wiley, New York, 1983.
4. Ronald G. Havelock, *Planning for Innovation*, Institute for Social Research, Ann Arbor, Michigan, 1970.
5. Claude E. Shannon and Warren Weaver, *Mathematical Theory of Communication*, University of Illinois Press, Urbana, IL., 1949.
6. Human Interaction Research Institute, *Putting Knowledge to Use*, National Institute of Mental Health, Rockville, MD, 1976.

7. *Knowledge Creation, Diffusion, Utilization, 1* (1) (1978).
8. Center for Libraries and Education Improvement, *Alliance for Excellence*, U.S. Department of Education, Washington D.C., 1984.
9. Edward T. Hall, *The Silent Language*, Doubleday, New York, 1959.
10. Commission on Higher Education and Adult Learners, *Adult Learners: Key to the Nation's Future*, American Council on Education, Columbia, MD, 1984.
11. Edward L. Thorndike, *Adult Learning*, Macmillan, New York, 1928.
12. Dorothy Field and Sylvia Weishaus, *Consistancy of Personal and Intellectural Characteristics Over Half a Century*, ERIC DOC 240 447, Educational Resources Information Center, Colombus, Ohio, 1983.
13. Ronald Gross, *Independent Scholar's Handbook*, Addison-Wesley, Reading, MA, 1982.
14. Herbert C. Morton and Anne J. Price, "View on Publications, Computers, Libraries." *Scholarly Commun.*, 5:1–16 (Summer 1986).
15. Theodore Roszak, *Cult of Information: Folklore of Computers and the True Art of Thinking*, New York, Pantheon Books, 1986.
16. Alvin Toffler, *The Third Wave*, William Morrow, New York, 1980.
17. Ernest R. Hilgard and Gordon E. Bower, *Theories of Learning*, 4th ed., Prentice-Hall, Englewood Cliffs, NJ, 1975.
18. D. W. Massaro, *Experimental Psychology and Information Processing*, Rand McNally, Chicago, IL, 1975.
19. R. L. Solso, ed., *Information Processing and Cognition: Loyola Symposium*, Lawrence Erlbaum, Hillsdale, NJ, 1975.
20. C. K. West and Stephen Foster, *Psychology of Human Learning and Instruction*, Wadsworth, Belmont, CA, 1976.
21. Steven F. Foster, "Ten Prinicples of Learning Revised in Accordance with Cognitive Psychology." *Ed. Psychol.*, *21*(3): 235–243 (Summer 1986).
22. Patricia Kirby, *Cognitive Style, Learning Style, and Transfer Skill Acquisition.* National Center for Research in Vocational Education, Columbus, OH, 1979.
23. Herman A. Witkin and H. B. Lewis, *Personality Through Perception*, Harper, New York, 1954.
24. Samuel Messick, et al., *Individuality in Learning*, Jossey-Bass, San Francisco, CA, 1976.
25. Herman A. Witkin, et al. *Field Dependent and Field Independent Cognitive Styles*, Educational Testing Service, Princeton, NJ, 1974.
26. Laurent A. Dolz, *Effective Teaching and Mentoring*, Jossey-Bass, San Francisco, 1986.
27. Malcolm S. Knowles, *Using Learning Contracts*, Jossey-Bass, San Francisco, CA, 1986.
28. J. A. Fodor, *Representations: Philosophical Essays on the Foundations of Cognitive Science*, MIT Press, Cambridge, MA, 1981.
29. George Spear and Donald Mocker, "Organizing Circumstances," *Adult Ed. Q.*, *35*(1)1–10 (1984).
30. Allen Tough, *Adult's Learning Projects*, Ontario Institute for Studies in Education, Toronto, Canada, 1971.
31. Malcolm Knowles, *Self-Directed Learning*, Association Press, New York, 1975.
32. Fritz Heider, *Psychology of Interpersonal Relations*, Wiley, New York, 1958.
33. Philip C. Kendall and S. B. Hollon, eds., *Cognitive Behavioral*

Interventions: Theory Research and Procedures, Academic Press, New York, 1979.
34. John Nisbet and Janet Shucksmith, "Seventh Sense," *Scottish Ed. Rev.*, *16*(2): 75–87 (November 1984).
35. Albert Bandura, *Social Learning Theory*, General Corporation, Morristown, NH, 1977.
36. Kurt Lewin, *Field Theory in Social Science*, Harper, New York, 1951.
37. Bergen R. Bugelski, *Priniciples of Learning and Memory*, Praeger, New York, 1979.
38. Richard E. Mayer, *Thinking, Problem Solving, Cognition*, Freeman, New York, 1983.
39. William C. Knaak, *Learning Styles*, National Center for Research in Vocational Education, Columbus, OH, 1983.
40. Laura P. Weisel, *Adult Learning Problems*, National Center for Research in Vocational Education, Columbus, OH, 1980.
41. Roger C. Schank and Robert Abelson, *Scripts, Plans, Goals and Understanding*, Erlbaum, Hillsdale, NJ, 1977.
42. Daniel Bobrow and Allan Collins, *Representation and Understanding*, Academic Press, New York, 1975.
43. Margaret A. Boden, *Artificial Intelligence and Natural Man*, Basic Books, New York, 1977.
44. Stanley J. Rimers, *Process Model: Subconscious Artificial Intelligence*, Syndetic Corp., Omaha, NB, 1986.
45. John R. Anderson, *Cognitive Psychology and Its Implications*, Freeman, San Francisco, CA, 1980.
46. Robert J. Sternberg, ed., *Handbook of Human Intelligence*, Cambridge University Press, New York, 1985.
47. K. J. Gilhooly, *Thinking: Directed, Undirected, and Creative*, Academic Press, New York, 1982.
48. Ian I. Mitroff and Ralph H. Kilmann, *Methodological Approaches to Social Science*, Jossey-Bass, San Francisco, CA, 1978.
49. Dedre Gentner and Albert Stevens, *Mental Models*, Lawrence Erlbaum, Hillsdale, NJ, 1983.
50. K. A. Ericsson and Herbert A. Simon, *Protocol Analysis: Verbal Reports as Data*, MIT Press, Cambridge, MA, 1984.
51. John Peters, *Adult Problem Solving and Learning*, ERIC DOC 200 758, American Educational Research Association, Los Angeles, April 1981.
52. Robert J. Sternberg and Douglas K. Detterman, *What is Intelligence?* Ablex, Norwood, NJ, 1986.
53. Helga A. Rowe, *Problem Solving and Intelligence*, Lawrence Erlbaum, Hillsdale, NJ, 1985.
54. John R. Anderson, *Architecture of Cognition*, Harvard University Press, Cambridge, MA, 1983.
55. Theodore M. Shlechter and Michael P. Toglia, eds., *New Directions in Cognitive Science*, Ablex, Norwood, NJ, 1985.
56. Zenon W. Pylyshyn, *Computation and Cognition*, MIT Press, Cambridge, MA, 1984.
57. Thomas Sowell, *Knowledge and Decisions*, Basic Books, New York, 1981.
58. Lloyd D. Noppe and J. M. Gallagher, "Cognitive Style Approach to Creative Thought" *J. Personality Assessment*, *41*: 85–90 (1977).
59. Amos Tversky and Daniel Kahneman, "Framing of Decisions and the Psychology of Choice," *Science*, *211*: 453–458, January 30, 1981.

60. Paul A. Kolers and Henry Roediger, "Procedures of Mind." *J. Verbal Learning and Behav.*, *23*(4): 425–449 (August 1984).
61. John W. Slocum, "Cognitive Style in Learning and Problem Solving" in *Experiences in Management and Organizational Behavior*, (Douglas Hall, ed.), Wiley, New York, 1982.
62. Kerry A. Johnson and Marilyn D. White, "Cognitive Style of Reference Librarians." *RQ*, *21*(3): 239–246 (Spring 1982).
63. Aidan P. Moran, "Unresolved Issues in Research on Field Dependence-Independence." *Soc. Behav. Personality*, *13*(2): 119–125 (1985).
64. Gregory Ashby and James T. Townsend, "Varieties of Perceptual Independence," *Psychol. Rev.*, *93*(2), 154–179 (1986).
65. V. R. Griffin, "Self-Directed Learning: Theories." *International Encyclopedia of Education*, *8*: 4517–4519, Pergamon Press, New York, 1985.
66. Alan B. Knox, *Adult Development and Learning*, Jossey-Bass, San Francisco, CA, 1977.
67. Norman D. Kurland, *National Strategy for Lifelone Learning*. Postsecondary Education Convening Authority, Institute for Educational Leadership, Washington, D.C., 1976.
68. Norman V. Overly, *Model for Lifelong Learning*, Phi Delta Kappa, Bloomington, IN, 1980.
69. Marcie Boucovalas, *Interface: Lifelong Learning and Community Education*, School of Education, University of Virginia, Charlottesville, VA, 1979.
70. Warren Ziegler, "Concept of the Learning Stance," in *The Learning Stance*, Syracuse Research Corporation, Syracuse, NY, 1979.
71. George Spear, "Beyond Organizing Circumstances," in *Self-Directed Learning*, (Huey Long, ed.), Adult Education Department, University of Georgia, Athens, GA, 1987.
72. Allen Tough, "Major Learning Efforts." *Adult Ed.*, *28*: 250–263 (1978).
73. Roger Hiemstra, ed. *Policy Recommendations Related to Self-Directed Adult Learning*, Occasional Paper No. 1. Administrative and Adult Studies, Syracuse University, Syracuse, NY, 1980.
74. Huey B. Long, "In Search of a Theory of Adult Cognitive Development." *J. Res. and Dev. Ed.*, *13*(3): 1–10 (Spring 1980).
75. Joan E. Knapp, *Assessing Transfer Skills*, National Center for Research in Vocational Education, Columbus, OH, 1979.
76. K. Patricia Cross, *Adults as Learners*, Jossey-Bass, San Francisco, CA, 1981.
77. Patrick R. Penland, *Towards Self-Directed Learning Theory*, ERIC Document, ED 209 475, Columbus, OH, 1981.
78. Malcolm Knowles, *Modern Practice of Adult Education: Andragogy Versus Pedagogy*, Association Press, New York, 1977.
79. Robert M. Smith, *Learning How to Learn: Applied Theory for Adults*, Follett, Chicago, IL, 1982.
80. Huey B. Long, ed., "Piagetian Theory and Adult Learning." *J. Res. Dev. Ed.*, *13*(3): 1–77 (Spring 1980).
81. Stephen Brookfield, ed., *Self-Directed Learning: From Theory to Practice*, Jossey-Bass, San Francisco, 1985.
82. Rosemary S. Caffarella and Judy M. O'Donnell, "Self-Directed Learning A Critical Paradigm Revisted (unpublished manuscript); *Presentation Papers*, Commission of the Professors of Adult Education, Milwaukee, WI, November 1985.

83. Donald Mocker and George Spear, *Lifelong Learning: Formal, Informal and Self-Directed Learning*, ERIC Document, ED 220 723, 1982.
84. K. Patricia Cross, *Missing Link: Connecting Adult Learners to Learning Resources*, College Entrance Examination Board, New York, 1978.
85. Wilhelm Wundt, *Foundation of Psychological Psychology*, Breitkopf and Hartel, Leipzig, Germany, 1873.
86. Anne Eberle and Sandra Robinson, *Adult Illiterate Speaks Out* (Contract No. 400-79-0036), National Institutes of Education, Washington, D.C., 1980.
87. Mary J. Even, "Adult Learning Process." *Perspectives in Adult Learn-*
Linda J. Reisser, *Facilitation Process for Self-Directed Learning*,
88. Ph.D. Thesis, University of Massachusetts, 1973.
Maurice Gibbons, "Toward a Theory of Self-Directed Learning: A
89. Study of Experts Without Formal Training", *Journal of Humanistic Psychology*, *20*(2):41-56, Spring, 1980.
90. Nigel Ford, "Towards a Model of Library Learning in Educational Systems." *J. Librarianship*, *11*(4): 247–260 (October 1979).
91. Harlan Cleveland, *Knowledge Executive: Leadership in an Information Society*, E. P. Dutton, New York, 1985.
92. Patrick Wilson, *Second-Hand Knowledge: An Inquiry into Cognitive Authority*, Greenwood Press, Westport, CT, 1983.
93. Benjamin Bloom, *Taxonomy of Educational Objectives: Cognitive Domain*, McKay, New York, 1956.
94. David Krathwohl, *Taxonomy of Educational Objectives: Affective Domain*, McKay, New York, 1964.
95. Elizabeth Simpson, *Taxonomy of Educational Objectives for the Psychomotor Domain*, U.S. Office of Education, Washington D.C., 1976.
96. Sherry Turkle, *Second Self: Computers and the Human Spirit*, Simon and Schuster, New York, 1984.
97. Endel Tulving and G. H. Bower, "Logic of Memory Representation," in *Psychology of Learning and Motivation* (G. H. Bower, ed.), Academic Press, New York, 1974.
98. Patrick R. Penland, *Self-Directed Learning in America*, SLIS, University of Pittsburgh, Pittsburgh, PA, 1977.
99. Herbert A. Simon, *Sciences of the Artificial*, MIT Press, Cambridge, MA, 1969.
100. U. Neisser, *Cognition and Reality*, Freeman, San Francisco, CA, 1976.
101. W. H. Wicklegren, *Cognitive Psychology*, Prentice-Hall, Englewood Cliffs, NJ, 1979.
102. A. Binet, *Experimental Studies of Intelligence*, Schleicher, Paris, 1903.
103. K. Duncker, "On Problem Solving." *Psycho. Monogr.*, *58*(270) (1945).
104. Cuadra Associates, *Library and Information Science Research Agenda for the 1980s*, Office of Libraries and Learning Technologies, U.S. Department of Education. Washington, D.C., 1982.

PATRICK R. PENLAND

COBOL

BRIEF HISTORY OF THE LANGUAGE

Mainframe Cobol

In 1959 CODASYL (Conference on Data Systems Languages) was created to study the problem of business computer operations. The committee included representatives of U. S. government agencies (primarily the Department of Defense), computer manufacturers, universities, and computer users. CODASYL decided to create a new language with common elements for all computers. The language was to be understandable by people with very little computer training. The purposes to be met by the language follow:

1. Machine independence—basic language elements had to be common to all computers.
2. Source language—easy to understand and use.
3. Include business terminology or cater to business terminology.

The results of the CODASYL conference were produced in April 1960. They included the initial specifications for COBOL. As COBOL started to change, computer manufacturers recognized the importance of new capabilities, and each manufacturer took a different approach to implementation. The result was that early compilers suffered from a number of problems:

1. In general, only a small subset of the total language was implemented. Different compilers implemented different subsets, thus reducing compatibility.
2. Compilation times were long, especially if the compiler implemented a large subset of the total language, sometimes taking up to an hour to compile a single program.
3. Machine code was inefficient, requiring two to five times the memory space of an assembly language program written to perform the same functions.

It soon became apparent that if computer manufacturers were left to their own devices, COBOL would no longer be the common language intended by CODASYL. Consequently, in 1968, the American National Standards Institute (ANSI) published suggestions for making COBOL a standard programming language. The standards were to be used by computer manufacturers for the construction of COBOL compilers. Thus, COBOL compilers for mainframes that generally conform to the ANSI recommendations are referred to as Standard COBOL or ANSI COBOL compilers.

To ensure continuity of communication between users and suppliers of COBOL, ANSI has tried to encourage the standardization of the language. ANSI also ensures that the standards are generally recognized and accepted and represents United States interests in international standardization.

Because COBOL development is a continuing process, the COBOL Committee of CODASYL continually reviews and acts on proposals to update the COBOL language. In 1974, ANSI revised the specifications that were current at that time and developed a new set of standards. In 1978, the ANS COBOL Committee (X3J4) used the latest version of the *Journal of Development*, published by the COBOL Committee of CODASYL, and the 1974 standards as the basis for the latest revision. Although the standards were set in 1980, it has taken some time for the manufacturers to incorporate them into their compilers; therefore, the newest standards will be known as ANSI 1985 standards.

Microcomputer COBOL

Microcomputers became popular in the mid-1970s and, compared with mainframe computers, had little memory storage capacity; they were physically small enough and inexpensive enough to attract the "hobbyist." During the early stages of development, many users would write their programs in assembler language. However, BASIC, a language initially developed for use on time-sharing systems in colleges, was quickly established as the standard high level language for microcomputers. Although as a programming language BASIC suffered from lack of standardization across manufacturers, because several dialects of the language were (and are) actually in use, it enjoyed the important advantage of being easy to learn and fast to write, attracting the casual user. From the point of view of a business user, it had four major disadvantages, which limited its potential:

1. It had no business or pseudobusiness terminology, and the construction of the instruction set was more appropriate for scientific applications than for business applications.
2. Programs were often not easily transferrable from one type of computer to another.
3. Communication between machine and user tended to be of a question-and-answer nature. This, in part, reflected the early history of BASIC and its use in conjunction with "hard copy" teletype terminals installed in schools and colleges. Given that business users were more likely to use visual display units (VDUs), effective acceptable communication required the ability to format pseudodocuments on a screen, particularly for the input of data. Such an ability was beyond most versions of BASIC.
4. BASIC generally offered a limited range of file access and retrieval methods. Data files stored on direct access devices, such as floppy disks, involved devising algorithms that provided a pointer to the position of the required record relative to the beginning of the file. This was simple enough if records, such as those of customers, were numbered sequentially, for example, 1, 2, 3. In practice, file coding was rarely that simple, and often complex algorithms had to be constructed, especially when dealing with particular problems presented by record identifiers containing alphabetic characters.

To make microcomputers appealing to the business user, manufacturers had to find a common, business-oriented language already in existence. What better choice than COBOL? Would this choice be in tune with the typical microcomputer business user? The estimation was that the answer would be yes. The type of businesses most likely to consider the use of microcomputers are probably small concerns with limited budgets for data processing functions, where programming costs account for a substantial part of the total cost of the system.

In introducing a computer-based system, a small company looks for such benefits as reduced staffing levels, greater accuracy, and better statistical analysis. Such businesses could not be expected to change their methods of operation to adjust to an automated system. Typically, a business holds files of customers, supplies, products, and so forth, each of which has a unique identifier or "key." The coding system used to identify customers, supplies, or products is not the same from business to business, although some basic similarities do exist. Take the case of a shoe store where the code indicates the shoe style, shoe size, and shoe color: SA10BL could indicate a sandal, size ten, and blue in color. This code, used by the shoe store, would not necessarily be used by a TV supply company or a small hardware store. The programming language should accommodate the user's needs and not vice versa.

With this type of user in mind we might consider three reasons for putting COBOL on a microcomputer:

1. It is easier and less costly to find and hire a COBOL programmer than a BASIC programmer because there are more experienced COBOL programmers on the market, as they learned it from the mainframe.
2. Noncomputer personnel involved in system analysis and design find it easier to understand programming functions expressed in business terms, something that COBOL readily supplies.
3. Auditors have had to examine source code of programs to ensure that procedures are executed correctly. Normally they are accustomed to examining programs written in COBOL.

In addition to satisfying the above-mentioned "human" aspects of programming, COBOL provides sequential, relative, and indexed file organizations that enable the user to access records with complex keys in either a sequential or random fashion, as the application dictates. Finally, microcomputer COBOL has built-in capabilities that allow screen formatting, making it easier to store and retrieve business forms. Thus, COBOL provides a virtually complete programming system for the small business user who wishes to develop programs that can be easily used by VDU operators, whose main concern is to input data in a manner as easy as possible through manageable forms.

Versions of COBOL for microcomputers

The most popular versions of COBOL for microcomputers are listed as follows:

1. CIS COBOL by Microfocus. It has a software package (FORMS II) to facilitate the design and creation of pseudoforms on the screen (CRT monitor).

2. COBOL-80 by Microsoft. Used extensively for IBM Personal Computers and compatibles, it has a screen section to help with the design of formatted screen for interactive use of COBOL programs.
3. RM/COBOL by Ryan-McFarland Corporation. RM/COBOL was the first compiler to be developed and used in microcomputers.

The developers of COBOL compilers for microcomputers have used the 1974 ANS COBOL standards, making changes as needed to accommodate the package to the new environment. Unfortunately, there is no standardization of software for microcomputers, meaning that each of the COBOL compilers contains some "enhancements," such as the FORMS II of CIS COBOL and the Screen Section of the Data Division of COBOL-80. These features make the programs less portable than desired, contrary to one of the goals of COBOL, which originally was designed to be machine independent and portable.

Basic Hardware and Software Requirements

Minimum memory required for COBOL compilation is 64K of random access memory (RAM). Although an 80-column display screen is recommended, one can either use a monitor or a TV screen with an RF modulator. Communications via the Asynchronous Communications Adapter are accessible directly from COBOL; however, protocols must be handled by the user's programs.

Two 320K disk drives are sufficient for average users, although a hard disk with 10M or more of storage is preferred by heavy users as it is faster and allows them to store a large number of programs into just one unit.

A line printer is necessary to generate hard-copy reports. The choice between dot–matrix and letter quality is left up to the individual user.

The software package usually consists of a manual and two diskettes containing the COBOL compiler and library files. In addition, one should have an editor package to enter the source code.

COMPARISON WITH ANS COBOL

Micro COBOL versus Mainframe COBOL

The power and range of facilities provided by available versions of COBOL depend upon the size and power of the computer used. Essentially, microcomputer versions contain all the major file formats and instructions. However, certain variations of instructions are omitted. For example, MOVE, ADD, and SUBTRACT instructions do not support the CORRESPONDING option, and the REMAINDER option in a DIVIDE instruction is not provided.

Similarly, certain facilities that require major processing, such as sorting, merging, and report writing, are currently available in mainframe versions of COBOL only. These are not major drawbacks because report programs are relatively simple to write, and there are several special-purpose programs available to sort data files.

Thus, the business microcomputer user should view COBOL as one of several pieces of software to be utilized rather than as the sum total of software required.

After comparing the list of reserved words from both COBOL-80 and RM/COBOL to the 1974 ANS list of COBOL reserved words, the author found

80 and 110 words missing, respectively, from the microcomputer COBOL versions. The reason for the discrepancy is that COBOL-80 has implemented a subset of SORT command, whereas RM/COBOL has not.

Also to be noted is the fact that both COBOL-80 and RM/COBOL have introduced new words to handle interactive data entry and retrieval of records. From 22 to 40 words have been added as reserved words, most of which deal with screen-formatting features such as setting the background and foreground colors, automatically moving the cursor from line to line as data is entered, reversing the video, and beeping to alert the user when exceptional events occur.

Aside from the differences, COBOL compilers for microcomputers conform to the "low-intermediate" level of the ANS X3.23–1974, providing from 9 to 10 of the 12 functional processing modules present in standard COBOL:

Nucleus
Sequential I/O
Relative I/O
Indexed I/O
Library
Communication(*)
Interprogram communication
Table handling
Sort/merge(*)
Debugging
Report writer(*)
Segmentation

(*) Features not present in current microcomputer COBOL versions.

Each module consists of two levels, where level 1 is a subset of level 2. To qualify as a COBOL compiler, the package must provide for all the level 1 features of the Nucleus, Table handling, and Sequential I/O modules. The remaining nine modules can be implemented optionally. Microcomputer COBOL compilers include all of level 1 for each module implemented and most of level 2 for certain modules. Therefore, one should not be misled by the phrase "low-intermediate" because COBOL compilers pack a lot of flexibility.

What follows reproduces or paraphrases the characteristics of the 12 modules as implemented in the IBM Personal Computer (PC) COBOL:

The Nucleus consists of the basic elements needed to process data. It includes:

- All of level 1: ACCEPT, ADD, ALTER, COMPUTE, DISPLAY, DIVIDE (without REMAINDER), EXIT, GOTO, IF, INSPECT, MOVE, MULTIPLY, PERFORM, STOP, SUBTRACT
- Plus these features of level 2:
  - Conditions— Level 88 conditions with value series or range
    - Use of logical AND/OR/NOT
    - Use of algebraic relational symbols for equality or inequalities (=, >, <)
    - Implied subject, or both subject and relation, in relational conditions
    - Sign test
    - Nested IF statements; parentheses in conditions

Verbs— Extensions to ACCEPT and DISPLAY for formatted screen handling
Acceptance of data from DATE/DAY/TIME
STRING and UNSTRING statements
COMPUTE with multiple receiving fields
PERFORM...VARYING...UNTIL

Identifiers— Mnemonic names for accept or display devices
Procedure names consisting of digits only
Qualification of names (in Procedure Division statements only)

The Table handling module provides for the definition and manipulation of tables of contiguous items. Tables may be one, two, or three dimensional, and the elements in them may be accessed through one, two, or three subscripts or indices. Tables must be of fixed length. In IBM COBOL, this module includes

- All of level 1: OCCURS in the Data Division and subscripting capabilities in the Procedure Division
- Plus these features of level 2:
  - SEARCH statement (full format)
  - SET statement

The File-processing module provides for the definition of Sequential, Relative and Indexed I/O. In IBM COBOL this module includes

- All of level 1: CLOSE, DELETE, OPEN, READ, REWRITE, USE, WRITE
- Plus these features of level 2:
  - Reserve clause
  - Multiple operands in OPEN and CLOSE, with individual options per file
  - VALUE OF FILE-ID is data-name
  - Sequential I/O: EXTEND mode for OPEN
    WRITE ADVANCING data-name lines
    LINEAGE phrase
    AT END-OF-PAGE clause
  - Relative and Indexed I/O:
    Dynamic access mode (with READ NEXT)
    START (with key relations EQUAL, GREATER, OR NOT LESS)

The Sort/merge module, which provides for ordering and combining records contained in one or more files according to key values present within the records, is not supported by IBM COBOL. A reduced version of the SORT facility is present in RM/COBOL.

The Report writer module allows for the overlaying of Procedure Division object code during the execution of the program. Level 1 is supported in its entirety by IBM COBOL.

The Library module allows for the specification of text to be copied (inserted) into a source program by the compiler at compilation time. The text to be copied resides in either the system or the user's library. All of level 1 is supported by IBM COBOL.

The Debugging module provides the user with capabilities to monitor a data item or procedure during execution of the object program. IBM COBOL supports

Special extensions to ANSI-74 Standards, providing convenient trace–style debugging.
Conditional compilation: Lines with a D in column 7 are bypassed unless WITH DEBUGGING MODE is given in the SOURCE-COMPUTER paragraph.

The Communication module, which provides for accessing, processing, and creating messages used to communicate with local and remote terminals is not supported by IBM COBOL.

The Interprogram communication module provides the user with capabilities to transfer control from one program to another during a run. This facility allows both programs to access data items explicity identified by the programmer. IBM COBOL supports

- All of level 1: CALL, CHAIN, EXIT PROGRAM
- Plus: LINKAGE Section

Referring to the Nucleus and Table handling modules, IBM COBOL includes all level 2 features except

- General:
  - One cannot use figurative constant ALL for literals greater than one character
  - One cannot qualify names in the Environment Division
- Data Division:
  - OCCURS DEPENDING ON... is not supported
  - One cannot intermix a level-88 item containing a list of items with a range of items (either list or range may be used but not both at one time)
  - Binary data items always require 2 bytes:
    —PICTURE 9(5) only allows a range of -32768 to 32767
    —PICTUREs 9, 99, 999, and 9999 are equivalent to PIC 9(5) for binary items
    —An error message is given when more than five digits are specified
  - Unsigned binary data items:
    —PIC 9 is equivalent to PIC S9
  - RENAMES phrase is not supported
- Procedure Division:
  - MOVE, ADD, and SUBTRACT do not support CORRESPONDING
  - Multiple destinations for results of arithmetic statements are not supported
  - Division remainders are not provided
  - INSPECT in level 2 is not supported
  - Arithmetic expressions in conditions are not supported
  - Multiple index Keys are not supported
  - Special language for tape handling is not supported
  - Level 1 RERUN facility is not supported
  - Interprogram communication and LIBRARY modules are implemented to level 1 only.

COBOL and the Editor

Because the program is keyed in directly without help of keypunch or decks of cards, the user has to become acquainted with the program editor that is supplied by the manufacturer of the computer system being used. Editors let the programmer create and modify lines of COBOL code that will eventually become the source code input to the COBOL compiler. Although some editors will provide the user with special features, such as automatic line numbering and automatic tabbing to COBOL margin boundaries, this is not true in all cases; therefore, the user should consult the editor manual that will be used.

Generally, basic editing functions supported by most editors will allow the user to

1. Enter one line of COBOL source code at a time. Pressing the ENTER key will terminate the entry of a line of code and store it into memory.
2. Correct errors by positioning the cursor under the desired character to be changed and either deleting it, replacing it, or inserting extra characters between it and the previous character. The cursor can move to the left, right, top, and bottom of the screen by specially marked keys on the keyboard.
3. Add new lines of code at the end or insert them within already existing lines.
4. Delete lines of code.
5. Move sections of code from one area of the program to another.

Normally, if the programmer is working with a line editor, only one line of code can be modified at any one given time; this means that there will be no movement up and down the screen and the lines will be accessed by their numbers.

When the user has finished keying in the source program, it should be written to the disk by using the appropriate command supported by the system being used. After this phase of program creation is finished, the user should then exit the editor and return to the supervisor to continue with the compilation and execution of the program.

COBOL Structure and Organization

Every COBOL source program is divided into four divisions. Each division has a special function, must be placed in its proper sequence, and must begin with a division header. Program structure and order is shown below:

```
IDENTIFICATION DIVISION.
      PROGRAM-ID. program-name.
      [AUTHOR. comment-entry...]
      [INSTALLATION. comment-entry...]
      [DATE-WRITTEN. comment-entry...]
      [DATE-COMPILED. comment-entry...]
      [SECURITY. comment-entry...]
ENVIRONMENT DIVISION.
      [CONFIGURATION SECTION.
            [SOURCE-COMPUTER. entry]
```



```
            [OBJECT-COMPUTER. entry]
            [SPECIAL-NAMES. entry]]
      [INPUT-OUTPUT SECTION.
            [FILE-CONTROL. entry...
            [I-O-CONTROL. entry...]]
DATA DIVISION.
      [FILE SECTION.
            [file-description-entry
            record-description-entry...]...]
      [WORKING-STORAGE SECTION.
            [data-item-description-entry...]...]
      [LINKAGE SECTION.
            [data-item-description-entry...]...]
      [SCREEN SECTION.
            [screen-description-entry...]...]
PROCEDURE DIVISION.
      [DECLARATIVES.
            [section-name SECTION.
            [paragraph-name. [sentence]...]...]...
      END DECLARATIVES.]
            [[section-name SECTION. [segment number]]
            [paragraph-name. [sentence]...]...]...
```

Identification Division

This division is the smallest and simplest division of a COBOL program. As the name indicates, it supplies identifying data about the program. It has no effect on the execution of the program but is required as a means of identifying the program to the computer's operating system. Optionally, this division may also include the name of the author, the installation where the program was run, as well as the date the program was written and compiled.

Environment Division

One of the functions of this division is to specify the computer on which the source program will be compiled. However, its major function is to specify the data files required for input and output operations and associate each of them with a particular device.

This is an important function because in any program that references or produces data files, it is necessary to establish a link between those files and the actual physical devices where they are stored or upon which they will be recorded. In other words, each file that is to be used has to be named and the program has to be instructed, via a symbolic name, to which physical input or output unit each file has been assigned.

Data Division

This division is one of the longest of the first three divisions, which can be considered as preparatory divisions in a COBOL program. It provides a detailed description of all the data that the program is to input, process, generate, or output. To accomplish this, the Data Division is divided into sections. The File Section describes data that is contained in either input or output files. The Working–Storage Section describes user-defined constants and data that are developed internally and placed into intermediate

storage or into specific formats for output reporting purposes. The Screen Section, present in some microcomputer COBOL compilers, describes data used to format the screen so that it looks exactly like a preprinted fill-in-the-blanks form, familiar to all of us. Once the screen has been programmed, it may be used for both data entry and data retrieval.

Procedure Division

This division contains all the instructions necessary for the solution of a problem. This is where the programming logic is implemented and where the files, records, and data defined in the previous program divisions are manipulated to produce the desired results.

The modules that constitute the Procedure Division are written using meaningful English words, statements, sentences, and paragraphs.

Because COBOL is not completely machine independent, it should be mentioned just how much machine dependence there is on each division. The Identification Division is completely machine independent. The Environment Division is completely machine dependent. This was specifically designed to facilitate the conversion of a program from one computer system to another, so that changes would not have to be made throughout the program. When transporting a program from one computer to another, this division must be at least partially rewritten. The Data Division is transferrable if the data being described are compatible from microcomputer to microcomputer. The Procedure Division is machine independent.

Language Structure

COBOL resembles written English and, if used properly, is self-documenting. Because COBOL is a highly structured language, rules of punctuation and spelling are very important and should be learned quickly to avoid annoying errors. COBOL source code is made up of words, literals, special symbols, and delimiters.

Word Formation

COBOL words are of two types: reserved words and programmer-supplied names. Reserved words have preassigned meanings used in COBOL programs as key words, connectives, figurative constants, or special-character words. Because they have special meaning, they must be used as indicated in the COBOL manual. Figurative constants are used to name and reference specific constant values, such as SPACES or ZERO, whereas arithmetic operators (+, −, *, /, **) and relational characters (<, >, =) constitute the special-character words.

Programmer-supplied names are from 1 to 30 characters long and are used to identify data items, records, files, or procedures. The characters used in the programmer-supplied names are upper- and lowercase letters, hyphens, and digits. With the exception of procedure names, all other words must contain at least one letter or one hyphen; in other words, they cannot be all digits.

Statements, Sentences, Paragraphs, and Sections

Statements are instructions given to the computer. They are the basic executable entities that form the Procedure Division. Statements usually begin with a verb that indicates the action to be taken, followed by appropriate operands and other words needed for the completion of the statement.

There are three types of statements: imperative, conditional, and compiler directing.

An imperative statement specifies an unconditional action to be taken by the object program. For example,

```
MOVE 5.10 TO COST, or
WRITE STUDENT-RECORD AFTER ADVANCING 1 LINE, or
ADD 1 TO RECORD-COUNTER.
```

A conditional statement contains a condition that is tested to determine the path to be taken by the object code. Simple conditions include

1. Relational condition, which causes the comparison of two operands, such as

```
IF HOURS-WORKED > 40.0 PERFORM OVERTIME-MODULE ELSE PERFORM
STRAIGHT-TIME-MODULE.
```

2. Class condition, which determines if an operand is numeric or alphabetic, such as

```
IF STUDENT-NAME IS ALPHABETIC PERFORM DATA-ENTRY-MODULE.
```

3. Condition-name condition, which tests if a conditional variable is equal to a particular value or to a set or a range of values defined previously in the Data Division, such as

```
IF SENIOR PERFORM READY-TO-GRADUATE-MODULE.
```

4. Sign condition, which determines whether or not the algebraic value of a data item is less than, greater than, or equal to zero, such as

```
IF AMOUNT IS NEGATIVE PERFORM DEBIT-MODULE.
```

Compound conditions are formed by combining simple conditions with the logical operators AND, OR.

A compiler-directing statement causes the compiler to take a specific action during compilation, such as copying source code from a user's library. This is useful when large sections of identical code are used in many programs.

Sentences are either single statements or a series of statements ended by a period and followed by a space. Paragraphs are groups of sentences and must begin with a paragraph name. Sections are groups of paragraphs and must begin with a section header.

File Organization and Description

A COBOL file is a collection of records stored into and retrieved from diskettes (or hard disk). The language allows users to specify file organization, access method, and processing mode (defined with the OPEN statement).

Microcomputer COBOL supports three types of file organization: sequential, relative, and indexed; these logical structures are established when data files are created.

A sequential file is one in which the records are physically stored in the sequence in which they are written. Once written, the records are retrieved and processed in the same sequence. To access any particular record, the program would have to read all the preceding records from the beginning of the file.

A relative file is one in which the records are stored in specific positions relative to the beginning of the file. The record key (a field containing unique values) is used to indicate, directly or indirectly, the relative position of the record. There is no need to enter the records in any particular sequence as the system will store them automatically in their correct sequence. For example, a record with the value 5 for its record key is stored in the fifth location relative to the beginning of the file, regardless of whether or not records have been written to locations one through four.

An indexed file is one in which the records are stored and retrieved, based on the value of a unique record key contained within that record. An index is automatically maintained for each record and is the means by which records are accessed.

A file's access method determines the manner in which records are inserted, deleted, or modified. Sequential access is allowed for all three file organizations, and the processing order is based on that organization: For sequential organization, the order is that in which records were written to the file; for relative organization, the order is ascending, based on the relative record number of existing records; for indexed organization, the order is ascending, based on the value of the record keys being used. Random access is allowed only for relative and indexed files, and the processing order is specified by the user based on relative record number or specific record keys. Dynamic access is allowed only for relative and indexed files and permits the user to change between sequential and random accessing. Dynamic access allows random positioning into a file, followed by sequential processing.

The open mode of a file determines the actions to be performed on that file. The OPEN verb has several options:

1. OPEN INPUT file-name: The program will read (retrieve) records from that file.
2. OPEN OUTPUT file-name: The program will write (store) records into that file.
3. OPEN I-O file-name: The program will read and/or write records to that file.
4. OPEN EXTEND file-name: The program will append records to the end of that file.

When READ and/or WRITE commands are executed, COBOL provides for detection of errors or unusual conditions occurring during the processing of files, for example, AT END conditions in sequential files or INVALID KEY conditions in relative or indexed files. Files are described in the File Section of the Environment Division. This is done by means of an FD entry, which contains the clauses describing various characteristics of the file, such as the file's system label (if any), its external name, its blocking

Line Number   Source Line   IBM Personal Computer COBOL Compiler Version 1.00

```
       Identification Division.
       Program-id. Create Indexed File.
       Installation. University of Pittsburgh.
       Date-Written. April 10, 1986.
       Security. None.

      This program creates an indexed sequential inventory
      file.  The file is created interactively, and the user
      is prompted for data through a formatted screen.

       Environment Division.
       Configuration Section.
       Source-Computer. IBM-PC
       Object-Computer. IBM-PC
       Input-Output Section.
       File-Control.
       Select Index-data assign to disk
           file status is file-status-code
           access is sequential
           organization is indexed
           record key is part-num.
       Data Division.
       File Section.
       FD Index-data
           Label records are standard
           Value of file-id is "invent.dat"
           Data record is invent-rec.
       01  invent-rec.
           02 part-num              pic x(3).
           02 part-name             pic x(15).
           02 quant-on-hand         pic 999.
           02 unit-cost             pic 99V99.
       Working-Storage Section.
       01  work-area.
           02 reply                 pic x    value " ".
           02 file-status-code      pic xx   value " ".
           02 num-work              pic x(3).
           02 name-work             pic x(15).
           02 quant-work            pic 999.
           02 unit-work             pic 99V99.

       The first screen is used to either enter a new part number
       or to stop execution of the program by entering a 0.

       Screen Section.
       01  invent-screen1.
           02 line 3 column 2 value
                 'Enter part number or 0 (zero) to stop'.
           02 line 3 column 40 pic x(3) to num-work auto.
```

FIGURE 1.   Compiled listing of program to create indexed file. (continued)

The second screen collects the rest of the data.

```
01  invent-screen2.
    02 line 5 column 2 value
           'Part name'.
    02 line 5 column 20 pic x(15) to name-work auto.
    02 line 7 column 2 value
           'Quantity on hand'.
    02 line 7 column 20 pic 999 to quant-work auto.
    02 line 9 column 2 value
           'Unit cost'.
    02 line 9 column 20 pic 99V99 to unit-work auto.
Procedure Division.
```

This module controls the execution of the rest of the program. The tasks of opening and closing files are done in this module.

```
001-Main-Module.
    Open output Index-data.
    Display (1, 1) erase.
    Display invent-screen1.
    Accept invent-screen1.
    If num-work not = "0"
            display invent-screen2
            accept invent-screen2
            perform 002-Detail-Module thru
            003-Exit-Detail until num-work = "0".
    Close Index-data.
    Stop Run.
```

This module accepts the information from the operator and stores it into the indexed sequential file.

```
002-Detail-Module.
    Move num-work to part-num.
    Move name-work to part-name.
    Move quant-work to quant-on-hand.
    Move unit-work to unit-cost.
```

If a problem occurs while storing a record, the program displays the error code and stops the program's execution.

```
    Write invent-rec
            invalid key display "Error in loading"
            display "File status code:" file-status-code
            move "0" to num-work.
    If num-work not = "0"
            display (1, 1) erase
            display invent-screen1
            accept invent-screen1.
```

FIGURE 1. (continued)


```
    If num-work not = "0"
            display invent-screen2
            accept invent-screen2.
003-Exit-Detail.
    Exit.
```

FIGURE 1. (continued)

a

```
Enter part number or 0 (zero) to stop    ...
Part name    ...............
Quantity on hand    ...
Unit cost    .....
```

b

```
Enter part number or 0 (zero) to stop    123
Part name    Beach Bags
Quantity on hand    010
Unit cost    07.90
```

FIGURE 2. (a) Empty formatted screen. (b) Formatted screen after data entry.

```
123Beach Bags     0100790 145Beach Chairs    0172138 216Towels
0350982 275Life Jackets    0202799 322 Goggles    0150525 331Snorkels
0301119 486Flippers    0121579 561Beach Balls    0050387 697Umbrellas
0181410 702Inner Tubes    0080643
```

FIGURE 3. Contents of indexed file (invent.dat).

```
Identification Division.
Program-id. Maintain Indexed File.
Installation. University of Pittsburgh
Date-Written. May 5, 1986.
Security. None.
```

This program maintains a previously created indexed sequential file. By means of a menu, it allows a user to (1) add records (2) delete records, (3) update records, (4) retrieve records.

```
Environment Division.
Configuration Section.
Source-Computer. IBM-PC.
Object-Computer. IBM-PC.
Input-Output Section.
File-Control.
Select Index-data assign to disk
    file status is file-status-code
    access is dynamic
    organization is indexed
    record key is part-num.
Data Division.
File Section.
FD Index-data
    Label records are standard
    Value of file-id is "invent.dat"
    Data record in invent-rec.
01  invent-rec.
    02 part-num                 pic x(3).
    02 part-name                pic x(15).
    02 quant-on-hand            pic 999.
    02 unit-cost                pic 99V99.
Working-Storage Section.
01  work-area.
    02 reply                    pic x    value " ".
    02 file-status-code         pic xx   value " ".
    02 num-work                 pic x(3).
    02 name-work                pic x(15).
    02 quant-work               pic 999.
    02 unit-work                pic 99V99.
    02 choice-code              pic 9.
         88      zero           value 0.
         88      one            value 1.
         88      two            value 2.
         88      three          value 3.
         88      four           value 4.
         88      five           value 5.
    02 error-flag               pic 9    value 0.
Screen Section.
```

This screen presents the menu to the user and accepts the chosen number.

FIGURE 4. Compiled listing of program to maintain the indexed file.


```
01  menu-screen.
    02 line 3 column 2 value 'This program will:'.
    02 line 5 column 10 value
         'Add records to the file if you enter a 1'.
    02 line 6 column 10 value
         'Delete records from the file if you enter a 2'.
    02 line 7 column 10 value
         'Update records in the file if you enter a 3'.
    02 line 8 column 10 value
         'Retrieve records from the file if you enter a 4'.
    02 line 9 column 10 value
         'Stop execution of the program if you enter a 5'.
    02 line 12 column 2 value
         'Please enter the number of your choice:'.
    02 line 12 column 50 pic 9 to choice-code auto.
```

This screen allows the user to enter information about a new record or to modify information in an already existing record The action is determined by whichever module is accessed during execution.

```
01  add-update-screen.
    02 line 3 column 2 value
         'Enter part number'.
    02 line 3 column 20 pic x(3) to num-work auto.
    02 line 5 column 2 value
         'Part name'.
    02 line 5 column 20 pic x(15) to name-work auto.
    02 line 7 column 2 value
         'Quantity on hand'.
    02 line 7 column 20 pic 999 to quant-work auto.
    02 line 9 column 2 value
         'Unit cost'.
    02 line 9 column 20 pic 99V99 to unit-work auto.
```

This screen shows record information requested by the user when the Retrieve option is selected.

```
01  retrieve-screen.
    02 line 3 column 2 value
         'The information you requested is:'.
    02 line 5 column 10 value
         'Part number'.
    02 line 5 column 30 pic x(3) from part-num auto.
    02 line 7 column 10 value
         'Part name'.
    02 line 7 column 30 pic x(15) from part-name auto.
    02 line 9 column 10 value
         'Quantity on hand'.
    02 line 9 column 30 pic 999 from quant-on-hand auto.
    02 line 11 column 10 value
```

FIGURE 4. (continued)

```
            'Unit cost'.
        02 line 11 column 30 pic 99V99 from unit-cost auto.
        02 line 20 column 2 value
            'Enter a 0 (zero) when ready'.
        02 line 20 column 30 pic 9 to choice-code auto.

 This screen prompts the user for the record to be worked on.

    01  del-update-screen.
        02 line 3 column 2 value
            'Enter part number to be:'.
        02 line 4 column 25 value 'DELETED,'.
        02 line 5 column 25 value 'UPDATED, or'.
        02 line 6 column 25 value 'RETRIEVED'.
        02 line 8 column 2 value 'Part number'.
        02 line 8 column 15 pic x(3) to num-work auto.
 Procedure Division.

 This module controls the execution of the rest of the
 program. The tasks of opening and closing files are
 carried out in this section.

    001-Main-Module.
        Open I-O Index-data.
        Move 0 to choice-code.
        Perform 002-Menu-Module until five.
        Close Index-data.
        Stop run.

 This module is used to display the menu, accept the chosen
 number, and direct the program to the respective module
 for execution. It is accessed through the Main module.

    002-Menu-Module.
        Display (1, 1) erase.
        Move spaces to invent-rec.
        Display menu-screen.
        Accept menu-screen.
        If one perform 003-Add-Module
        else If two perform 004-Delete-Module
            else If three perform 005-Update-Module
                else If four perform 006-Retrieve-Module.

 This module is used to add new records to the indexed file.
 It is accessed through the Menu module.

    003-Add-Module.
        Display (1, 1) erase.
        Display add-update-screenl
        Accept add-update-screen.
        Move num-work to part-num.
```

FIGURE 4. (continued)

Line Number Source Line IBM Personal Computer COBOL Compiler Version 1.00

```
        Move name-work to part-name.
        Move quant-work to quant-on-hand.
        Move unit-work to unit-cost.
        Write invent-rec invalid key
            display "Unable to add record #" num-work
            display "File status code:" file-status-code
            move 5 to choice-code.

 This module is used to delete existing records from the
 indexed file. It is accessed through the Menu module.

    004-Delete-Module.
        Perform 007-Request-Module.
        If error-flag = 0 delete Index-data record.

 This module is used to update existing records in the
 indexed file. It is accessed through the Menu module.

    005-Update-Module.
        Perform 007-Request-Module.
        If error-flag = 0
            perform 009-Change-Module.

 This module retrieves information on existing records in
 the indexed file. It is accessed through the Menu module.

    006-Retrieve-Module.
        Perform 007-Request-Module.
        Display (1, 1) erase.
        Display retrieve-screen.
        Accept retrieve-screen.

 This module is used to ask for the part number of the
 record to be Retrieved, Deleted, or Updated. It is accessed
 through any of those three modules.

    007-Request-Module.
        Display (1, 1) erase.
        Display del-update-screen.
        Accept del-update-screen.
        Perform 008-Read-Module.

 This module reads records from the indexed file. If an error
 occurs when reading, the file status code is displayed and
 the program execution stops. It is accessed through the
 Request module.

    008-Read-Module.
        Move num-work to part-num.
        Read Index-data record key is part-num
        invalid key
```

FIGURE 4. (continued)

Line Number Source Line IBM Personal Computer COBOL Compiler Version 1.00

```
                display "Unable to read record # " num-work
                display "File status code: " file-status-code
                move 1 to error-flag
                move 5 to choice-code.

     This module is used to update values stored in existing
     records. It is accessed through the Update module.

     009-Change-Module.
         Display (1, 1) erase.
         Display "Enter changes in appropriate lines".
         Display add-update-screen.
         Accept add-update-screen!
         If name-work not = spaces
                move name-work to part-name.
         If quant-work not = 0
                move quant-work to quant-on-hand.
         If unit-work not = 0
                move unit-work to unit-cost.
         Rewrite invent-rec.
```

FIGURE 4. (continued)

factor, the character code used to represent its contents in secondary storage (CODE-SET), and its data record name(s). Because files are a collection of records, it is possible to have more than one record type within a file. Data records are described in a hierarchical manner, using level numbers to indicate the position of data items in the hierarchy. The record is considered to be a group item, and its fields are elementary items. In some cases, contiguous elementary items within a record may be combined into a group for joint access.

```
This program will:

    Add records to the file if you enter a 1
    Delete records from the file if you enter a 2
    Update records in the file if you enter a 4
    Retrieve records from the file if you enter a 5

Please enter the number of your choice:      0
```

FIGURE 5. Menu screen.

a

```
Enter part number     ...
Part name        ...............
Quantity on hand      ...
Unit cost      .....
```

b

```
Enter part number     280
Part name        Diving Masks
Quantity on hand      025
Unit cost      08.30
```

FIGURE 6. (a) Empty formatted screen to add records. (b) Formatted screen after data entry.

a

```
Enter changes in appropriate lines
Enter part number     ...
Part name        ...............
Quantity on hand      ...
Unit cost      .....
```

b

```
Enter changes in appropriate lines
Enter part number     331
Part name
Quantity on hand      027
Unit cost      .....
```

FIGURE 7. (a) Empty formatted screen to update records. (b) Formatted screen after update.

```
Enter part number to be

                        DELETED,
                        UPDATED, or
                        RETRIEVED

Part number     ...
```

FIGURE 8. Formatted screen to accept record number.

Sample Programs

The first sample program illustrates use of the Screen Section for interactive data entry. The compiled source code (Fig. 1), an empty formatted screen (Fig. 2a), and a filled-in formatted screen (Fig. 2b), as well as a dump of the created indexed data file (Fig. 3), are included.

The second sample program illustrates random access of an indexed sequential file. The program allows for addition of new recnrds, deletion, and/or update of existing records. The compiled source code (Fig. 4), the menu screen (Fig. 5), samples of empty and filled-in formatted screens for addition (Fig. 6a,b) and update (Fig. 7a,b) tasks, formatted screens for deletion (Fig. 8) and retrieval (Fig. 9), as well as a dump of the modified index data file (Fig. 10), are included. The first program was compiled and executed on an IBM PC and the second was compiled and executed on a COMPAQ.

```
The information you requested is
    Part number        331
    Part name      Snorkels
    Quantity on hand      027
    Unit cost      11.19
Enter a 0 (zero) when ready 0
```

FIGURE 9. Formatted screen showing retrieved information.


```
  123Beach Bags     0100790 145Beach Chairs     0172138 216Towels
0350982 275Life Jackets    0202799 322Goggles     0150525 331Snorkels
  0271119 486Flippers     0121579 561Beach Balls     0050387 697Umbrellas
     0181410 702Inner Tubes        0080643 280Diving Masks     0250830
```

FIGURE 10. Contents of indexed file after maintenance (invent.dat).

BIBLIOGRAPHY

The following references were used in writing this article:

Atkinson, W. J., Jr., and Pa. A DeSanctis, *IBM PC COBOL*, Reston Publishing Co., Inc., Reston, Va., 1985.

Carver, K., *Structured COBOL for Microcomputers*, Brooks/Cole, Ca., 1983.

Cowden, J. A., "PC Reviews IBM COBOL," *PC Mag., pp. 96–102 (October 1982)*.

Stang, N., *COBOL for Micros*, Newnes Technical Books, London, 1983.

IBM Personal Computer, Computer Language Series, *COBOL Compiler*, Microsoft, Boca Raton, Fl., 1982.

IDA M. FLYNN

CODING GRAY LEVEL AND BINARY IMAGE DATA: IMAGE DATA COMPRESSION

1. INTRODUCTION

The research of image data compression dates back to the early 1920s, when there was an effort to transmit digital pictures over the Bartlane cable picture-transmission systems between New York and London. Pictures were coded in five distinct brightness levels. The transmission time was reduced from more than 1 week to less than 3 hours [1].

As the result of digital computer development in the 1960s, the field of digital image data compression has experienced vigorous growth. It attracted most attention around the 1970s. During that time, the research interests were concentrated on some fundamental problems, such as image fidelity, statistical image models, predictive coding, and optimal transforms for transform coding.

From the late 1970s until now there has been continuing interest in image data compression. Some new compression methods, for example, block truncation, vector quantization (VQ), synthetic-high systems, and motion-compensated techniques, have been proposed. Concepts of image processing and pattern recognition have been incorporated into image data compression, and more sophisticated coding schemes have appeared. More and more research is conducted to investigate methods for object-oriented binary image coding with applications toward pictorial information systems and computer vision systems.

In addition to applications of image data transmissions, such as digital TV and facsimile, image data compression is very closely related to image analysis, pattern recognition, and knowledge-based systems. Image feature extraction and description can also be considered as a sort of data compression. It compresses image data into feature vector or symbolic description, which reflects the characteristics of the objects presenting in the image for the purpose of recognition and interpretation. In various pictorial information systems, coding of the images not only saves the storage space but also facilitates efficient retrieval.

Data compression of gray-level images and binary images can be very different. Encoding of gray-level images is generally represented by the block diagram in Figure 1. The original image data are spatially correlated. Much redundacy exists. The mapping process maps image data into coefficients that are much less correlated. In other words, the coefficients are in a more compact form. The mechanism and form of mapping for different compression methods can be totally different. For example, the mapping of predictive coding is known as predictor. Transform coding uses orthogonal transforms as mapping. The quantizer subdivides the coefficients into smaller numbers of possible values that are suitable for coding. The

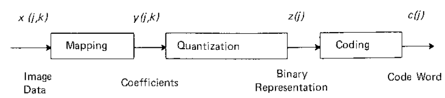


FIGURE 1 Block diagram of gray-level image coding.

coder assigns code words to the output of the quantizer. There are extensive discussions about coder design techniques in information theory and communications theory; therefore, we will not discuss the topic in this article. From a methodology point of view, gray-level image data compression methods are distinguished by the approaches of mapping and quantization. Section 2 is devoted to gray-level image data compression. After a brief discussion of image data models. two basic approaches, digital predictive coding method (DPCM) and transform coding, are dealt with in some detail. New approaches, such as VQ, synthetic-high method, and motion-compensated coding, are also introduced later.

It is well known that facsimile is one of the important applications of binary image data compression. On the other hand, binary image data compression is most applicable to pictorial data bases, which are kernel modules for pictorial information systems, office automation systems, computer-aided design (CAD) systems, computer vision systems, and so forth. Section 3 discusses oridinary binary image data compression techniques. Section 4 is dedicated to object-oriented binary image coding, with its aim toward pictorial data management. Three coding methods, chain coding, object-oriented run-length coding, and quadtree (mainly linear quadtree) will be described in detail.

2. GRAY-LEVEL IMAGE DATA COMPRESSION

2.1 Basic Concepts

Before detailing techniques for gray-level image data compression, this subsection describes three basic concepts: image fidelity, mathematical models of image data, and optimal quantization.

2.1.1 Image Fidelity

For the error-tolerance compression system discussed in this section, image fidelity is an important measure for performance of a compression system. Commonly used objective image fidelity measures are mean square error (MSE) between input image and output image and MSE signal-to-noise ratio (SNR) of output image.

Let $f(j,k)$ denote a digital image with $j,k = 0,1,\ldots,N-1$; the output image of a compression system is denoted by $g(j,k)$. The MSE between $f(j,k)$ and $g(j,k)$ is then defined as

$$\mathrm{MSE} = \frac{1}{N^2}\sum_{j=0}^{N-1}\sum_{k=0}^{N-1}[g(j,k) - f(j,k)]^2 \tag{1}$$

The MSE SNR of output image is defined as

$$\mathrm{SNR} = \frac{\sum_{j=0}^{N-1} \sum_{k=0}^{N-1} [g(j,k)]^2}{\sum_{j=0}^{N-1} \sum_{k=0}^{N-1} [g(j,k) - f(j,k)]^2}$$

The objective measures are easy to compute and, therefore, are often used for compression system design and tuning. The objective image fidelity measures MSE and SNR are image independent, although they sometimes do not reflect the actual visual quality of an image. Quite often, output images of a compression system with the same MSR appear to have different visual qualities due to the characteristics of the human visual system. Images are viewed by people. The subjective measure is therefore a final judgement to an image. The subjective measure of an image can be evaluated by a number of people by showing tham an image and averaging their evaluations. To evaluate the performance of a compression system, one may present the input and output images to a number of observers to evaluate the differences.

2.1.2 Image Models

Mathematically modeling images is a fundamental problem for image data compression. The Gaussian– Markov field model has been proved as a good candidate in the sense of feasibility and effectiveness. It is assumed an image is a sample picture from a two-dimensional discrete homogeneous Gaussian–Markov field described in [Ref. 2].

$$f(j,k) = \sum_{m=0}^{a} \sum_{n=0}^{b} \theta_{mn} f(j - m, k - n) + u(j,k) \quad m = n \neq 0 \tag{3}$$

where

(a) $E[u(j,k)\ f(j - m, k - n)] = 0 \quad 0 \leqslant m \leqslant a,\ 0 \leqslant n \leqslant b,\ m = n \neq 0$

(b) $E[u(j,k)\ u(j,k)] = \sigma^2 \delta_{jl} \delta_{kl}$

The two conditions state that u(j,k) is a zero mean independent Gaussian field noise. In equation (3), θ is called the regression coefficient. Figure 2 shows the region of summation.

A special case of the Gaussian–Markov field model, with a and b equal to 1, is usually called first-order Gaussian–Markov field.

$$f(j,k) = \theta_1 f(j - 1,k) + \theta_2 f(j - 1,k - 1) + \theta_3 f(j,k - 1) + u(j,k) \tag{4}$$

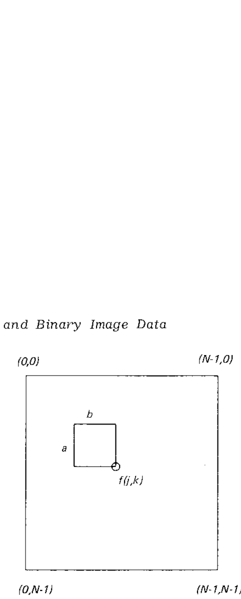


FIGURE 2 Region of summation for Gaussian–Markov field model.

Equation (4) says that the gray level of pixel (j,k) is spatially related to its neighbor pixels (j − 1,k), (j − 1,k − 1), and (j, k − 1). The degree of this gray-level dependence is described by coefficients θ's. For a given image, one can fit this model to the image data, solve for parameters θ_1, θ_2, θ_3, and noise variance σ^2. Delp showed good fitting results to several natural images and good DPCM coding results based upon this model [3].

In the model defined by Eq. (3), the gray level of a pixel is related only to its upward and left neighbors. Extending the dependence to its other neighbor pixels is an obvious procedure.

It is worth noting that the Gaussian–Markov model defined by Eqs. (3) and (4) is actually a low-pass filter model. This model, therefore, will reflect low frequency and random characteristics of image data. Consequently, the compression methods based on this model will have no problem as far as low frequency signal is concerned, Special techniques should be investigated to compensate for the high frequency performance of the system.

2.1.3 Optimal Quantization

Quantization is used to represent input data by a limited number of discrete values (usually integers). The input data are supposed to be randomly distributed with the distribution function p(x). S set of thresholds are then optimally chosen to convert input data into digital numbers with minimum error. In Figure 3, $x_1, x_1, \ldots, x_{N+1}$ denotes the set of thresholds, and and $y_1, y_2, \ldots y_N$ denotes output numbers.

If the MSE is assumed as the performance measure for the quantizer, then we have

$$e = \sum_{i=1}^{N} \int_{x_i}^{x_{i+1}} [y_i - x]^2 p(x)dx \tag{5}$$

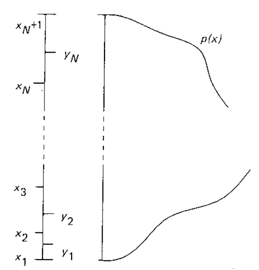


FIGURE 3 Input data distribution and quantization.

The solution of Eq. (5) with minimum error e [4] is

$$x_{i+1} = \frac{1}{2}(y_i + y_{i+1}) \tag{6}$$

$$\int_{x_i}^{x_{i+1}} (x - y_i)p(x)dx = 0 \tag{7}$$

Obviously for a given distribution density function and the number of output digital values (word length), one can obtain a set of thresholds $x_1, x_2, \ldots, x_{N+1}$ by solving Eqs. (6) and (7), using the iterative technique. Bearing in mind that the optimization here is subject to the valid distribution density function and in the sense of MSE. If the density function does not fit actual input data, the quantization error will not be guaranteed to be minimum.

2.2 Predictive Coding

2.2.1 Basic Concepts

From the image model defined in Eq. (4) the gray level of a pixel can be estimated from that of its neighbor pixels if parameters θ_1, θ_2, θ_3 are known. Following this idea, the predictive coding system is constructed. Figure 4 shows the block diagram of the basic DPCM system.

One can drop the noise term and rewrite Eq. (4) as

$$f(j,k) = \theta_1 f(j-1,k) + \theta_2 f(j-1,k-1) + \theta_2 f(j,k-1) \tag{8}$$

θ_1, θ_2, and θ_3 can be estimated from a set of sample image data. Based on the above formula, the predictor estimates the gray level of the current

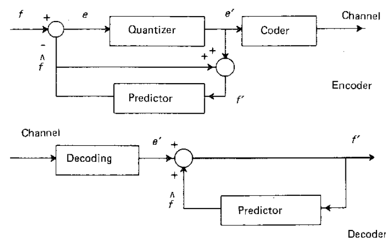


FIGURE 4 Block diagram of the basic DPCM system.

pixel from that of previous three neighbor pixels. The estimation is compared with the actual value to produce an estimation error. The predictor and comparator hare work together as the mapping device in Figure 1. It maps original image data into error data. The error data are much less correlated and have smaller dynamic range; therefore, the data require less bits for quantization. The encoder codes the quantized data and sends for transmission or storage.

In the receiver side, data are first decoded and then added to the estimated data to construct the original image. The predictor in the receiver side is a duplication of that in the transmitter side.

The basic predictive coding system in Figure 4 is based on the Gaussian–Markov model. In practice, image data quite often violate this model. Large prediction errors occur when abrupt edge, for example, is encountered. This will increase the dynamic range of prediction error and lead to large distortion.

Fortunately, the development of image processing and pattern recognition provides us with many effective techniques for image texture detection and classification. Based on these techniques, adaptive image-coding systems can be designed. For the different types of textures, the system adaptively selects a corresponding predictor or quantizer. This yields the concepts of adaptive prediction and adaptive quantization, which are to be described in the following sections.

2.2.2 Adaptive Quantization

The idea of adaptive quantization is to use several different quantizers, each fitting a certain type of image texture. The design of an adaptive quantization DPCM system involves two basic problems: One is to detect and classify image texture; the other is to design a set of quantizers fitting to the corresponding set of textures with minimum error.

The design of better adaptive quantization DPCM systems needs thorough understanding of texture analysis and edge-detection techniques in the image-processing field. However, there have been investiga-

tions into the possible texture activity functions for quantizer control [5]. Schafer proposed a texture activity function as follows [5]:

$$A_{MD} = \underset{i,j \varepsilon DN}{\mathrm{MAX}} |d_{ij}| \tag{9}$$

where $d_{ij} = f_j' - f_i'$

$$A_{WD} = \underset{i,j \varepsilon DN}{\mathrm{MAX}} \left\{ |\delta_i| + \frac{1}{4} (\delta_i + |\delta_j|) [1 - \mathrm{sign}\ (e_0)] \right\} \tag{10}$$

where $\delta_i = f_i' - f_0$. The sign function is defined as

$$\mathrm{sign}\ (e_0) = \begin{cases} 1 & e_0 > T \\ 0 & |e_0| \leqslant T \\ -1 & e_0 < -T \end{cases}$$

DN denotes a neighborhood, a set of neighbor pixels of current pixel, f_0. The structure of neighborhood of f_0 is shown in Figure 5.

The final texture activity function A_{MWD} is calculated by

$$A_{MWD} = \mathrm{MAX}\ [A_{MD}\ ,\ A_{WD}] \tag{11}$$

It is obvious that A_{MD} describes the maximum difference among neighbor pixels, whereas A_{WD} describes the maximum difference between the current pixel and its neighbor pixels. The texture activity function A_{MWD} reflects the possible maximum gray-level changes occurring within the neighborhood of the current pixel.

The texture activity function A_{MWD} is then used as a control parameter to adaptively select L separate quantizers according to

$$Q = \begin{cases} Q_1 & A_{MWD} < a_2 \\ Q_2 & a_2 \leqslant A_{MWD} \leqslant a_3 \\ \cdot & \\ \cdot & \\ \cdot & \\ Q_L & a_L \leqslant A_{MWD} \end{cases} \tag{12}$$

The threshold set $a_2, a_3, \ldots, a_L$ is chosen empirically. It depends on the characteristics of the image to deal with and on the method to design quantizers.

In Section 2.1.3, it was shown that a quantizer is specified by two parameters: One is the output word length; the other is, the distribution density function of input data. If Gaussian distribution is assumed for all cases, the mean and deviation are two parameters to specify the

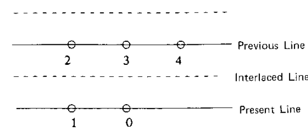


FIGURE 5 Neighborhood structures for texture classification.

distribution function. Because of the presence of active textures in images, images in practice tend to violate the Gussian–Markov model. The statistical distribution function of the prediction error based on this model cannot be considered simply as Gaussian. Intuitively, we can consider it as a composition of several Gaussian distributions, each specified by its own mean and deviation, corresponding to one type of textures. For example, image area with low activity measure has low deviation for its prediction error, whereas area with abrupt edges tends to have large deviation. One should keep in mind that human eyes are very sensitive to edges. Special attention should be paid to preserve edge information. To improve fidelity of predictive coded images, two possible methods can be used to design adaptive quantizers.

One possible way is to design a set of quantizers, $Q_1, Q_2, \ldots, Q_c$, with the same output word length. For each quantizer Q_i, the mean or deviation is different from that of others. The quantization thresholds of quantizers then turn out to be quite different from each other. Table 1 is an example of the thresholds of an adaptive quantization DPCM system. One can see from the table that the larger the texture activity measure is, the larger the quantization steps are.

Alternatively we can design a set of quantizers with different output word lengths. More bits are assigned to more active image area, and less bits to less active area. This scheme emphasizes active image textures, which are more important to the human visual system.

The bit rate can be reduced further using adaptive quantization by about 0.5 to 1 bit per pixel, compared with ordinary DPCM coding schemes.

TABLE 1 Thresholds of an Adaptive Quantizer

Thresholds	Texture Activity Measure
0, 3, 8, 15, 24, 25	$A_{MWD} < 15$
0, 7, 14, 23, 34, 47	$15 \leqslant A_{MWD} < 35$
0, 11, 22, 35, 48, 65	$35 \leqslant A_{MWD} < 100$
0, 15, 30, 45, 64, 85	$100 \leqslant A_{MWD}$

2.2.3 Adaptive Prediction

Adaptive quantization uses a fixed predictor, allows variation of dynamic ranges of prediction errors, and adaptively quantizes the prediction errors. On the contrary, adaptive prediction attempts to minimize prediction errors by adapting the prediction to the local image textures. One fixed quantizer can be used in this case because of the low prediction error. Adaptive prediction and motion-compensated estimation (to be described in Section 2.6.1) are based on the same mechanism, attempting to reduce the bit rate by keeping the prediction error as low as possible.

Control of adaptive prediction is more complicated then that of adaptive quantization, where one needs to care only about the prediction error. For adaptive prediction, one should understand under what circumstances a certain amount of prediction error is produced and then carefully design texture models by adjusting the prediction parameters in order to keep the prediction error as low as possible. As mentioned previously, the gray level of the current pixel is predicted by the weighted sum of its neighbor pixels. These weights are determined by a certain image model. So far we have used the Gaussian–Markov model. To reduce the prediction error, the additional image local texture model should be designed to fit the actual image texture. The basic procedures for adaptive prediction are first to detect and classify local textures and then to design a predictor to fit the detected texture. There have been some papers reporting research on adaptive prediction based on edge detection. Here we review the work by Zhang [6], who classified the textures of the neighborhood depicted in Figure 6 into four classes, characterized by

$$\hat{f}_0 = \begin{cases} h_1 & \text{flat area} \\ h_2 & \text{horizontal contour} \\ h_3 & \text{straight line other than horizontal} \\ h_4 & \text{texture} \end{cases}$$

The corresponding four predictors are defined by the following equation.

$$h_1 = \frac{5}{8} f_1' + \frac{1}{8} (f_6' + f_7' + f_8') \tag{13-1}$$

$$h_2 = \frac{3}{4} f_1' + \frac{1}{4} f_7' \tag{13-2}$$

$$h_3 = \frac{1}{4} f_{k-1}' + \frac{1}{2} f_k' + f_{k+1}' \qquad k \in \{6,7,8,9\} \tag{13-3}$$

$$h_4 = \frac{1}{5} (f_5' + f_6' + f_7' + f_8' + f_9') \tag{13-4}$$

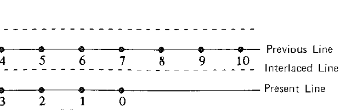


FIGURE 6 Prediction and edge-detection structure.

The rules to identify texture type of the neighborhood are as follows (the gray-level range is assumed to be 256).

1. The neighborhood is said to be flat if

$$\text{MAX } \{|d_{12}|, |d_{15}|, |d_{16}|, |d_{17}|\} < 20 \tag{14}$$

   The corresponding prediction Eq. (13-1) is an ordinary one, except more weight is on the left pixel.
2. A horizontal edge is assumed if

$$\text{MAX } \{|d_{12}|, |d_{23}|\} < \text{MIN } \{|d_{15}|, |d_{16}|, |d_{17}|, |d_{18}|\} \tag{15}$$

   Equation (15) implies that there is not much gray-level change in the current line, but a considerable change occurs along the vertical direction. Vertical change represents the horizontal edge. The prediction of f_0 in this case depends very much on its left pixel. This is reflected in Eq. (13-2).
3. The edge direction other than horizontal is identified by

$$\text{MIN } \{|d_{1,k-1}|\} < 51 \qquad k = 6,7,8,9 \tag{16-1}$$

$$\text{sign } (d_{12}) \text{ sign } (d_{k-1,k-2}) = 1 \tag{16-2}$$

   The above two equations state the conditions of a nonhorizontal edge existence as (a) there is not much gray-level change in the vertical direction; (b) at the location of the detected edge, the signs of gray-level changes of two neighboring lines have to be identical. On the other hand, in order to avoid the noise influence, the sign function is set to zero for small argument values, for example, < 7.

   As usual, more weights are put to neighbor pixels having less differences referring to the current pixel. In Eq. (13-3), k is the subscript in MIN $|d_{1,k-1}|$.
4. Texture is identified by rapid gray-level changes, which are detected by

$$\text{sign } (d_{12}) \text{ sign } (d_{k,k-1}) = -1 \qquad k = 6,7 \tag{17}$$

$$\text{sign}\,(d_{k-1,k-2}) \neq \text{sign}\,(d_{k,k-1}) + \text{sign}\,(d_{k+1,k}) \tag{17-2}$$

The first equation is to verify that the directions of gray-level changes on two adjacent lines are not the same. The second equation further assumes that even in the same line, the directions of gray-level changes are different. The corresponding prediction in Eq. (13-4) is simply the average of 5 pixels.

Using the adaptive prediction described above, very promising results were obtained [6].

2.2.4 Remarks on Predictive Coding

Because of the mechanism of predictive coding, the system error and channel error are cumulative to reconstructed images. When error occurs, one can eliminate it only by reinitialing the prediction; otherwise, the error will be effective until the beginning of the next line.

Predictive coding is easy to implement in real TV time. There has been comprehensive research conducted in this area, most of which showed good results with 2 bits per pixel. Further research involves time-varying image coding.

2.3 Transform Coding

Transform coding uses transform as a mapping tool to map original image data into transform coefficients. Unlike DPCM coding, which maps image data pixel by pixel, transform coding usually performs two-dimensional transform on image data blocks. Different numbers of bits are then assigned to quantize each transform coefficient by using a bit allocation matrix. Some small coefficients can even be discarded. Because the transform coefficients are nearly uncorrelated, the quantization error and channel error will be distributed to all pixels within the image block through the inverse transform appearing as additive random noise. Transforms used for transform coding are capable of compacting the energy of the image block into a few transform coefficients; more than half have very small amplitudes and can be discarded before quantization. This results in a high data compression ratio.

In the following sections, we review cosine transform briefly, the one most suitable for data compression, and then discuss the adaptive transform coding method using texture analysis technique.

2.3.1 Cosine Transform

Although many transforms can be used for transform coding, for example, Fourier transform, Hadamard transform, Haar transform, and Slant transform, we chose to discuss cosine transform because it is known to be approximately optimal when MSE criterion and the Gaussian–Markov field model are assumed. Two-dimensional forward and inverse cosine transform is defined as

$$F(u,v) = \frac{4c(u)\,c(v)}{N^2} \sum_{j=0}^{N-1} \sum_{k=0}^{N-1} f(j,k)\ \text{con}\ \frac{(2j+1)\,u\pi}{2N} \times \text{con}\ \frac{(2k+1)\,v\pi}{2N} \tag{18-1}$$

$$f(j,k) = \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} c(u)\,c(v)\,F(u,v)\ \text{con}\ \frac{(2j+1)\,u\pi}{2N} \times \text{con}\ \frac{(2k+1)\,v\pi}{2N} \tag{18-2}$$

where

$$c(u),\ c(v) = \begin{cases} \sqrt{2}/2 & u,v = 0 \\ 1 & u,v = 1,2\ldots,N-1 \\ 0 & \text{otherwise} \end{cases}$$

It is well known that the Karhunen–Loeve transform, the optimal transform, is defined by the eigenvectors of the covariance matrix of the data to be transformed. If the first-order Markov sequence model of source data is assumed, it covariance matrix will have the form of the Toepliz matrix, as follows:

$$\begin{vmatrix} 1 & \rho & & \rho^{M-1} \\ \rho & 1 & \cdots & \rho^{M-2} \\ \cdot & & & \cdot \\ \cdot & & & \cdot \\ \cdot & & & \cdot \\ \rho^{M-1} & \rho^{M-2} & \cdots & 1 \end{vmatrix} \qquad 0 < \rho < 1$$

It is shown that the plot of the eigenvector of the Toepliz matrix for ρ = 0.9 and the plot of one-dimensional cosine transform basis functions resemble each other closely. Recall that we have Gaussian–Markov image data model assumption for image coding. Based on this model, cosine transform is the best approximation to K–L transform. K–L transform has no fast algorithms. Fast cosine transform is even faster than fast Fourier transform. Figure 7 shows several curves representing the variances of transform coefficients of different transforms. Apparently K–L transform and cosine transform are the most capable of compacting "energy."

When cosine transform is used for transform coding, an input image is first divided into subimages (image blocks); the block size is usually taken to be 16×16 or 8×8. After transformation on these image blocks, a

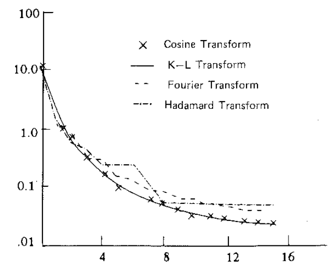


FIGURE 7 Variances of transform coefficients, $M = 16$, $\rho = 0.95$.

certain bit allocation matrix is designed to indicate the number of bits assigned to pixels in the block for their quantization. The bit allocation matrix is stored both in the transmitter side and the receiver side. According to the bit allocation matrix, the receiver can separate the coding data for pixels and insert zeros in the position of discarded pixels. The final reconstructed image is obtained by performing inverse cosine transforms on blocks and combining blocks together.

2.3.2 Adaptive Cosine Transform Coding

The transform is a fixed mapping tool in an image-coding system, and no adaptive procedures can be made upon it. Therefore, adaptive transform coding implies adaptive quantization only. Because image data are transformed block by block, adaptive quantization is also implemented blockwise by switching between a set of bit allocation matrices.

An adaptive transform coding system was proposed by Wu and Burge by introducing the texture analysis technique [7]. A more sophisticated method is in Ref. *8*. A block diagram of Wu and Burge's scheme is shown in Figure 8. An input image is divided into image blocks, each with 16×16 pixels, and cosine transform is carried out on each block. In transform domain, the blocks are classified into several classes and then normalized and quantized by using the corresponding normalization matrices and bit allocation matrices, respectively. These two matrices, once determined, are common to all images the system is to deal with and are stored in both the encoder and the decoder. Bookkeeping information is inserted in front of each block to indicate the class to which the block belongs.

Blocks in the same class are quantized with the same bit allocation matrix. To minimize the quantization distortion, the differences between the blocks assigned to the same class must be minimized. In other words, inner-class differences should be reduced as far as possible. The problem of block classification can be dealt with by pattern-recognition techniques

$$\mathrm{FIN} = \frac{\int_{\rho_1}^{\rho_2} d\rho \int_0^{\pi/2} F(\rho, \theta)\,d\theta \Big/ \int_{\rho_1}^{\rho_2} d\rho \int_0^{\pi/2} d\theta}{\int_{\rho_3}^{\rho_4} d\rho \int_0^{\pi/2} F(\rho, \theta)\,d\theta \quad \int_{\rho_3}^{\rho_4} d\rho \int_{\rho_0}^{\pi/2} d\theta} \tag{22}$$

Based on the texture measures above, a simple classifier is designed to classify transform patterns into 10 classes (Table 2). Classes 1–4 are nondirectional and non-high-frequency patterns, each having different texture activities. Classes 5–6 are patterns with horizontal directionality. Classes 7–8 are vertical directional. Classes 9–10 are high-frequency patterns. There are bit allocation matrices, each associated to one class. Bit allocation matrices are designed by training procedures. Figure 9 shows two bit allocation matrices corresponding to classes 3 and 6, respectively.

Good results were reported for house images using 0.26 bit per pixel [7].

2.4 Synthetic High Coding Systems

It is well known that in audio systems two or more channel amplifiers are used to gain sound fidelity. The same principle applies for image-coding systems. A block diagram of a synthetic-high image-coding system is shown in Figure 10.

An original image is decomposed into two components by a low-pass filter and a high-pass filter. The low-pass component gives the general view of the image without sharp gray-level changes. The high-pass

TABLE 2 Decision Function of Classifier

Class 1	MACE $<$ m 11	d 1 $<$ DIR $<$ d 2	
Class 2	m 11 $\leqslant$ MACE $<$ m 12	d 1 $<$ DIR $<$ d 2	
Class 3	m 12 $\leqslant$ MACE $<$ m 13	d 1 $<$ DIR $<$ d 2	FIN $>$ f1
Class 4	m 13 $\leqslant$ MACE	d 1 $<$ DIR $<$ d 2	FIN $>$ f1
Class 5	MACE $<$ m 21	DIR $\leqslant$ d 1	
Class 6	m 21 $\leqslant$ MACE	DIR $\leqslant$ d 1	
Class 7	MACE $<$ m 21	d 2 $\leqslant$ DIR	
Class 8	m 21 $\leqslant$ MACE	d 2 $\leqslant$ DIR	
Class 9	m 12 $\leqslant$ MACE $<$ m 13	d 1 $<$ DIR $<$ d 2	FIN $>$ f1
Class 10	m 13 $\leqslant$ MACE	d 1 $<$ DIR $<$ d 2	FIN $>$ f1

8654332211000000	8655444444333322
6543322110000000	4443332221110000
5443322110000000	2221110000000000
4332221100000000	0000000000000000
3332221100000000	0000000000000000
3332211000000000	0000000000000000
3222110000000000	0000000000000000
2221110000000000	0000000000000000
2211000000000000	0000000000000000
1110000000000000	0000000000000000
0000000000000000	0000000000000000
0000000000000000	0000000000000000
0000000000000000	0000000000000000
0000000000000000	0000000000000000
(a) 0.719 bpp	*(b)* 0.398 bpp

FIGURE 9 Example of bit allocation matrices.

filter component contains information of edges and detailed textures. In this section, three synthetic-high coding methods are described: block truncation coding, Laplasian pyramid method, and contour-texture approach.

The principle of synthetic high coding coincides well with the properties of the human visual system. It permits a considerable amount of redundancy reduction and shows promise for further research interests.

2.4.1 Block-Truncation Coding

Block-truncation coding [2, 9] divides an entire image of $N \times N$ into blocks of $M \times M$. The block size M is usually 4. Block-truncation coding can be considered as a synthetic-high system. The low-pass component is an image block with a constant gray level, the mean of original image data within the block. A bit plane indicating pixel gray level above or below the mean can be considered as a high-frequency component. To quantize image blocks, mean and variance are first computed.

$$m = \frac{1}{M \times M} \sum_{j=1}^{M} \sum_{k=1}^{M} f(j,k) \tag{23}$$

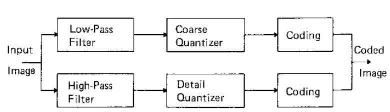


FIGURE 10 Block diagram of synthetic-high coding system.

$$\sigma^2 = \frac{1}{M \times M} \sum_{j=1}^{M} \sum_{k-1}^{M} f^2(j,k) - m^2 \tag{24}$$

Mean m is the average brightness of the block, and variance σ represents the texture activities. Mean and variance vary from block to block. One bit for each pixel is used to indicate whether the gray level is above or below the mean. The 1's and 0's indicate different reconstruction of the original image. It is specified by

$$Y_0 = m - \sigma\sqrt{q/p} \tag{25}$$

$$Y_1 = m + \sigma\sqrt{q/p} \tag{26}$$

where q and p are the number of pixels above and below the sample mean, respectively. Pixels coded with 1's are set to Y_1, and the others are set to Y_0.

The advantage of block-truncation coding is its simplicity. Good results can be obtained with 1–2 bits per pixel. For example, if the block size is 4×4, 8 bits are used to quantize the mean and variance; 16 bits for a 4×4 bit plane. The total bit rate is 2 bits per pixel.

2.4.2 Laplasian Pyramid Method

The Laplasian pyramid method intelligently uses pyramid data structure in image data compression. Starting from the original image $f_0(j,k)$ of size $N \times N$, a low-pass filter, which is actually a local averaging, is used to compute the low-pass component $f_1(j,k)$. The high-pass component $g_0(j,k)$ at level 0 is then obtained by

$$g_0(j,k) = f_0(j,k) - f_1(j,k) \tag{27}$$

Because f_1 is a low-pass component, it may be encoded at a reduced sample rate. If we reduce f_1 by a sample interval of 2, an image f_1 with smaller size $(N/2) \times (N/2)$ is obtained. An iteration procedure is performed on f_1 to get an image pyramid f_0, f_1, $\cdots$ and an error pyramid g_0, g_1, $\cdots$. Suppose a 5×5 average window is used, a filtering process to construct higher level image is defined by

$$f_l(j,k) = \sum_{m=-2}^{2} \sum_{n=-2}^{2} w(m,n)\, f_{l-1}(2j + m, 2K + n) \tag{28}$$

where w(m,n) is a weighting function.

Figure 11 shows a block diagram of a three-level Laplasian pyramid image-coding scheme. The original image f_0 is decomposed into two high-frequency components, g_0, g_1, and a low-pass component, f_2. We code f_2, g_0, and g_1 instead of f_0. f_2 has low resolution and requires less bits

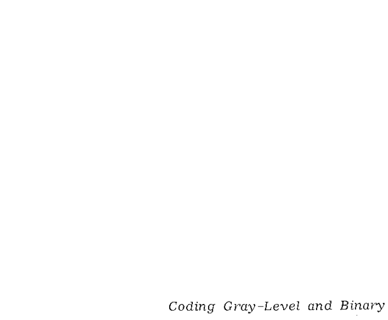


FIGURE 11 Block diagram of a three-level Laplasian pyramid image coding scheme.

for its coding. Error image g_0 and g_1 contain information of rapid edges and fine textures usually appearing within a small portion of the image area. Therefore, most pixels having error data close to zero can be neglected. This reduces the bit range a great deal. Good results were reported for head-and-shoulder images with bit rates of 1.58 and 0.73 bit per pixel.

The Laplasian pyramid coding scheme particularly suits progressive image transmission, which is required in pictorial information systems. For progressive image transmission, the image pyramid is organized and transmitted from top to bottom. In the receiver side, coarse image data are first received and reconstructed. The observer can view the image from coarse to fine; transmission may be terminated as soon as the contents of the image are recognized to be of no interest. This saves a lot of transmission time, for one may find an image of interest after several attempts.

2.4.3 Contour–Texture Scheme

Both block-truncation coding and the Laplasian pyramid method code images in terms of square image blocks. In practice, a square block does not fit the actual image texture. Large coding errors may result if there are abrupt edges and flat areas within one block. The contour–texture scheme presumes that images consist of objects, the area of which consists of homogeneous property. In the contour–texture scheme, the high-frequency components are contours of detected object regions, whereas low-frequency components are contexts of the regions. The contour–texture scheme consists of three processing steps: image segmentation by region growing, contour coding, and texture coding [*10*].

Region growing is a technique of image segmentation, which is operated on the original image to identify object regions. Because of fine structures and noise existence, the segmentation process may result in too many small regions, which increase the system complexity and make it impractical. Therefore, preprocessing, using an edge-preserving smoothing operation, is intended to reduce the local granularity of the original image without affecting its significant contours. Region-growing operations are performed on preprocessed images. Regions enclosed by contours are expected.

There are many techniques available for region growing and contour coding, which are described extensively in image-processing textbooks [e.g., *11*]. Contours can be encoded either by a sequence of approximated line and circle segments or by a sequence of contour points without approximation.

Texture coding here means coding of textures within object regions, which appear as object surfaces. As a matter of fact, there is not much texture left after preprocessing and segmentation because of the homogeneous property of the object region. Coding object surfaces can be achieved by fitting two-dimensional polynomials to regions. The parameters of polynomials are coded.

To reconstruct the original image, a contour image is first obtained from coded contour data. Surfaces represented by polynomials are then added to the contour image to get the final results. The results can usually preserve object shapes well, but the natural appearance of the original image has been lost in most cases: The object surface is too smooth. Sometimes, artificial "salt-and-pepper" noises are added to make the reconstructed image more natural.

The data compression ratio of the contour–texture scheme can reach as high as 50 for a camera image. Of course, there are still difficulties with this technique. First, a segmentation method does not exist that can be effective with a variety of images and can generate a limited number of object regions suitable to the coding scheme. The difficulties are partially due to the nature of images. Some images inherently have fine textures with rapid gray-level changes as their characteristics, which must not be filtered out by preprocessing.

2.5 Vector Quantization

2.5.1 Basic Concepts of VQ

It has been seen that the performance of image data compression depends highly on the method of quantization. VQ is an "intelligent" quantization technique, which utilizes some established methodologies in the field of pattern recognition and data base systems.

Figure 12 shows a block diagram of the VQ coding system. The name is relative to scalar quantization in Section 2.1.3, which quantizes data samples individually. When scalar quantization is used, one should invoke a mapping process to make use of data correlation in order to achieve data compression. VQ combines the two processes, decorrelation and quantization, into a single matching (retrieving in the decoder side) process.

It is assumed that we are dealing with p-dimensional vector $\vec{x} = (x_1, x_2, \ldots, x_p)$ and that each element in vector $\vec{x}$ is in digital form and represented by L digits. There are M possible vectors in a vector set X. The subscripts denote the sequential number of elements in a vector. Superscripts denote the vector membership number in a set of vectors.

Suppose we plot all of these vectors in a p-dimensional pattern space and use the pattern-recognition method to partition the space into N subspaces, each corresponding to a class. For each class of vectors, a prototype vector $\vec{y}$ is found such that the average distance from $\vec{y}$ to all vectors within this class is minimum. Obviously the dimension of $\vec{y}$ is the same as $\vec{x}$. Now we take the set of prototypes $Y = \{\vec{y}^1, \vec{y}^2, \ldots, \vec{y}^N\}$ as codebook, and store it in both encoder and decoder sides. In encoder,

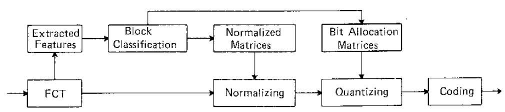


FIGURE 8 Block diagram of an adaptive transform coding system.

in cosine transform domain. The design procedure may include investigation of transform patterns, determination of class categories, design of texture measures which can characterize each class, and finding the decision functions by training procedures.

Image textures are well reflected in their cosine transforms. The cosine transform texture patterns can be characterized by directionality, frequency, and amount of texture activity.

The texture activity measure is mathematically described as "AC" energy.

$$\mathrm{MACE}(p,q) = \sum_{u=0}^{p} \sum_{v=0}^{p} F^2(u,v) - \sum_{u=0}^{q} \sum_{v=0}^{q} F^2(u,v) \tag{19}$$

where $q < p$. For computational simplicity, Eq. (19) can be rewritten as

$$\mathrm{MACE}\ (p,q) = \sum_{u=0}^{p} \sum_{v=0}^{p} |F(u,v)| - \sum_{u=0}^{q} \sum_{v=0}^{q} |F(u,v)| \tag{20}$$

The texture direction measure DIR is defined as the weighted average of angles of each transform component with respect to the u coordinate

$$\mathrm{DIR} = \frac{\sum_{u=0}^{M} \sum_{v=0}^{M} \tan^{-1}(\frac{u}{v}) |F(u,v)|}{\sum_{u=0}^{M} \sum_{v=0}^{M} |F(u,v)|} \tag{21}$$

The range of DIR is 0 to $\pi/2$. For example, the cosine transform pattern of an image block with vertical strips appears as a line next to the u axis; its DIR is close to zero.

The frequency measure is characterized by the slope of the cosine transform spectrum. Two frequency bands are specified. The average amplitude within the two specified frequency bands is computed; the ratio is defined as the measure.

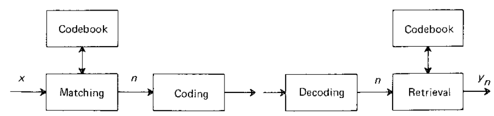


FIGURE 12 Block diagram of Vector (block) quantization.

for each input vector $\vec{x}$, the matching module finds the best match between $\vec{x}$ and $\vec{y}^n$, and outputs class number n. In decoder side, the class number n is used to retrieve $\vec{y}^n$, which is then thought to be the representative of $\vec{x}$ to reconstruct the original image. Intuitively, if $N << M$, VQ can achieve very high data compression ratio. The design of VQ system now is focused on the following problems: For a given error rate, find the minimum number of prototypes N (bit rate is $\log_2 N$), or for a given prototype number N, find the best set of prototypes with the minimum error rate. This codebook design problem can be solved by "training procedure" in pattern recognition. Fortunately, there are some methods available. Among them K-mean clustering algorithm now is most frequently used in VQ. It can be summarized as follows:

K-Mean Algorithm
Input: a set of input vector, X; expected number of classes, N.
Output: N prototype set Y.

1. Randomly choose N initial prototypes from X.
2. For each input vector $\vec{x}$, assign $\vec{x}$ to class n if $\vec{x}$ is closest to $\vec{y}^n$.
3. Update Y by taking $\vec{y}^n$ as the centroid of class n.
4. Check the changes of Y in this iteration. If changes are within a threshold ε, stop, and output Y; otherwise go back to step 2.

K-mean algorithm does not require any knowledge of source data statistics. It assumes the same deviation and a different mean for classes. If the situation violates the assumption, of course, K-mean algorithm will not give good results.

The matching process in Figure 12 can be modeled as a minimum distance classifier:

$$\text{Output } n, \text{ if } d(\vec{x}, \vec{y}^n) \leq d(\vec{x}, \vec{y}^m) \qquad \text{for all } m \neq n,\ m = 1,2,\ldots,N$$

In practice, if the dimension of the vector is considerably large, N can be too large to complete the decision-making procedure above with a reasonable computation cost. The techniques of indexing and retrieval in information systems should then be invoked.

Chang and his student Wang studied VQ on a set of test images. A compression ratio of around 10 was obtained without noticeable distortion [*12*].

2.5.2 Edge-Oriented VQ Image Coding

Ramamurthi and Gersho proposed an edge-oriented VQ image-coding scheme [*13*]. The vector (block) dimension was taken to be 16 (4x4). For edge/shade classification, two sets of gradient tables are formed, each for the horizontal (x) and vertical (y) directions. Let

$$h_{jk} = |f(j,k+1) - f(j,k)| \tag{29}$$

$$s_{jk} = \operatorname{sign}\,[f(j,k+1) - f(j,k)] \tag{30}$$

$$m_{jk} = [f(j,k+1) + f(j,k)]\ /\ 2 \tag{31}$$

The element of y table (matrix), y_{jk} is

$$y_{jk} = \begin{cases} s_{jk} & \text{if } h_{jk} > T_e\, m_{jk} \\ 0 & \text{otherwise} \end{cases}$$

where T_e is an edge-detection threshold (a typical value for T_e is 0.2). The x table can be similarly computed for a pair of pixels $f(j+1,k)$ and $f(j,k)$. At the same time, two counters are maintained for the x and y directions. The counter is incremented if $h_{jk} > T_s m_{jk}$, where T_s is another threshold and the typical value is 0.025.

Using the measures above, input vectors are preclassified. A vector is deemed to be a shade vector, if both the counters are separately less than a threshold (2 out of 16). A vector is an edge vector if there are more than a minimum number (e.g., 2) of +1's and -1's in either table. Otherwise, it is a mid-range vector. Edge vectors are further classified into 28 classes using these gradient tables. Each class has its different edge orientation and location.

Each class of vectors has its own sub-codebook, which is designed using training procedures. For a typical image, about 70–80% of vectors are mid-range classes. The matching complexity should be reduced for this class. This can be done by simplifying the distance measure MSE of two vectors. A discrete cosine transform is performed on two vectors. The distance measure of these two vectors is then defined as the differences of several low-frequency components. This truncation does not affect the result due to the distance preserving property of cosine transform. In the encoder, the six low-frequency components were stored in the codebook. An input vector is first classified and then transformed in real time, and a reduced complexity matching yield the optimum code vector. The decoder, of course, uses the original spatial domain codebook for reconstruction of the original image.

Using the technique above Ramamurthi and Gersho built codebooks for 0.7 bpp and 0.8 bpp. Both worked well for head-and-shoulder images [*13*].

2.6 Image Sequence Coding

So far we have discussed data compression techniques for still images or individual image frames. Attention has focused on searching the schemes that can make the best use of the spatial correlation properties of image data in order to facilitate high data compression ratio or, in other words, to reduce data rate of transmission.

A large portion of images in the real world exist as image sequences. For instance, TV images are time-varying image sequences. Landsat MSS images and color images are spectrum-varying image sequences. In addition to two spatial coordinates, image sequence is considered as the third coordinate. The third coordinate is most likely time or spectrum. Data correlations also exist in the third coordinate. This section describes the techniques for image sequence coding.

In principle, techniques for spatial data compression, with certain extension, are applicable to image sequence coding. Three-dimensional DPCM, three-dimensional transform, and transform/DPCM hybrid-coding methods belong to this type. However, instead of this type of coding technique, we will discuss techniques peculiar to image sequences: motion-compensated estimation for TV images and color image coding.

2.6.1 Motion-Compensated TV Image Coding

Because of the requirement of real-time transmission of TV images, predictive coding is often preferable. Motion-compensated TV image coding is an adaptive coding in time domain and is based on the following coding strategies:

1. Segmenting each frame into two parts: (a) background, the contents of which are the same as the previous frame; (b) a moving part, which has changed from the previous frame.
2. Two types of moving-area information are transmitted: (a) addresses specifing the location of the pixels of the moving area and (b) information to update gray level of moving-area pixels, which is motion-compensated prediction error in the case we are dealing with.
3. Use the frame buffer to match the coder bit rate to the constant channel rate. Because the motion in a TV scene occurs randomly and in bursts, the amount of information about the moving area will be time dependent. This makes the coder bit rate time varying.

It is clear that to reduce the bit rate, one should develop a good motion-compensated prediction. There has been much research conducted for motion estimation [*14*]. Because of the requirement of real-time transmission of TV images, only relatively simple methods are considered. In the following sections, we first describe recursive displacement estimation and determine a motion-compensated prediction image data coding system based upon it; a displacement estimation is then briefly introduced by block matching.

Recursive Displacement Estimation: The recursive displacement estimation method was proposed by Netraveli and Robbins [*15*]. The

algorithm iterates on a pixel-by-pixel (or on a small block of pixels) basis, revising their displacement estimate at every moving-area pixel by the following update formula:

$$\hat{D}^i = \hat{D}^{i-1} + U^i \tag{32}$$

where $\hat{D}^{i-1}$ is a displacement estimate of ith iteration and U^i is an update of $\hat{D}^{i-1}$ to $\hat{D}^i$. In other words, U^i is an estimate of $D - \hat{D}^{i-1}$. Let x represent spatial location, $f(x, t - \tau)$ and $f(x,t)$ be the gray level of the two successive frames at the location x and time t. We are now able to define displaced frame difference DFD $(x, \hat{D}^{i-1})$ as

$$\mathrm{DFD}(x, \hat{D}^{i-1}) = f(x,t) - f(x - \hat{D}^{i-1}, t - \tau) \tag{33}$$

Because displacement estimate $\hat{D}^{i-1}$ can take the value of a fraction, the evaluation of Eq. (33) should use an interpolation process in the discrete two-dimensional spatial space. As defined, DFD will converge to zero as $\hat{D}^i$ converges to the actual displacement D. This can be achieved by recursive estimation. Assume the prediction of pixel at location x_a has a displacement of $\hat{D}^{i-1}$ and a gray level of $f(x_a - \hat{D}^{i-1}, t - \tau)$, and the prediction error is DFD $(x_a, \hat{D}^{i-1})$. In the next iteration, the displacement estimate $\hat{D}^i$ produces the estimate error DFD $(x_a, \hat{D}^i)$, and $|\mathrm{DFD}(x_a, \hat{D}^i)| \leqslant |\mathrm{DFD}(x_a, \hat{D}^{i-1})|$. The iteration formula can take the following form

$$\begin{aligned}\hat{D}^i &= \hat{D}^{i-1} - \frac{\varepsilon}{2} \nabla_D [\mathrm{DFD}(x_a, \hat{D}^{i-1})]^2 \\ &= \hat{D}^{i-1} - \varepsilon \mathrm{DFD}(x_a, \hat{D}^{i-1}) \nabla_D \mathrm{DFD}(x_a, \hat{D}^{i-1})\end{aligned} \tag{34}$$

where ε is a small positive constant, and ∇_D is the gradient with respect to displacement D. From Eq. (33) we have

$$\nabla_D [\mathrm{DFD}(x_a, \hat{D}^{i-1})] = \nabla f(x_a - \hat{D}^{i-1}, t - \tau) \tag{35}$$

∇ is the gradient with respect to x. This gives us

$$\hat{D}^i = \hat{D}^{i-1} - \varepsilon \mathrm{DFD}(x_a, \hat{D}^{i-1}) \nabla f(x_a - \hat{D}^{i-1}, t - \tau) \tag{36}$$

We notice, from Eq. (36), that the update term to old estimation $\hat{D}^{i-1}$ is a vector quantity. It is parallel to the direction of the spatial gradient of the image gray level. Its amplitude is proportional to the motion-compensated prediction error.

Iteration defined by Eq. (36) can be implemented without much difficulty. Knowing $\hat{D}^{i-1}$, $f(x_a, t)$ and $f(x_a, t - \tau)$, $f(x_a - \hat{D}^{i-1}, t - \tau)$ is estimated by interpolation for the nonintegral of $\hat{D}^{i-1}$. Linear interpolation is usually used for its simplicity.

The constant ε was chosen to be 1/1024 in Ref. *15*. A large ε yields a quick convergence but a noisy estimation, whereas a small ε allows a finer value of displacement to converge.

Several recursive algorithms have been proposed since the research of Netravali and Robbin [*15*]. For example, Cafforie and Rocca proposed an extended version of their former recursive algorithm [*16*].

Based on their recursive algorithm, Netravali and Robbin constructed a motion-compensated TV image-coding system [*15*]. A simplified block diagram of a motion-compensated coder is shown in Figure 13. The outer loop produces prediction error by subtracting motion-compensated prediction from a current input TV field. The motion-compensated prediction is computed first by the displacement estimator and then by the motion-compensated prediction.

Based on frame difference (FDIF), the segmenter controls the displacement estimator through switch A and the motion-compensated predictor through switch B in order to let them work only in the moving area. The transmission of the prediction error is also controlled by the segmenter through switch C. Moving-area classification is performed using the following formula:

(1)	$\mid \mathrm{FDIF}(x) \mid > T_2$	or
(2)	$\mid \mathrm{FDIF}(x) \mid > T_1$	and
	$\mid \mathrm{FDIF} \mid$ at a, b, c, or d $> T_1$	and
	$\mid \mathrm{FDIF} \mid$ at A, B, C, or D $> T_1$	

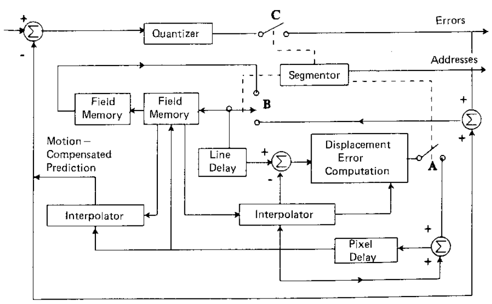


FIGURE 13 Block diagram of a motion-compensated coder.

The configuration of pixels in this formula is depicted in Figure 14. T_1 and T_2 are two thresholds, and $T_2 \geq T_1$. The typical values of T_1 and T_2 are 1 and 3 for the gray-level scale of 0–255, respectively.

As mentioned early in this section, the moving area is further classified into a compensable region, where motion-compensated prediction is accurate enough so that no update information needs be transmitted; and the uncompensable region, where the prediction error should be transmitted. The classification rule is as follows:

1. A pixel in a moving area is classified to be uncompensable it its amplitude of the motion-compensated prediction error is greater than 3 out of 255.
2. Between uncompensable pixels, compensable pixels having a run length less than or equal to 3 are also classified into uncompensable pixels.

Quantization of prediction error is a straightforward process. Coding of moving-area addresses should be designed carefully in order to reduce overhead information. Run-length coding of addresses was used in Ref. *15*.

By the motion-compensated prediction technique described above, Netravali and Robbins obtained 30–50% improvement for images of a speaking person and 22% improvement for images of complex and rapidly moving objects, compared with the coding without motion-compensated prediction.

Displacement Estimation by Block Matching: Block matching is a well-known technique in digital image processing. For displacement estimation purposes, an image block of size MxM with the center at the current pixel (j,k) is taken from frame i. The image block is then moved around in the previous frame i − 1, the correlation is computed between the block and its covered region in the previous frame i − 1. The correlation peak is found as the right displacement. For simplicity, the mean of the absolute frame difference (MAD) is proposed instead of the correlation.

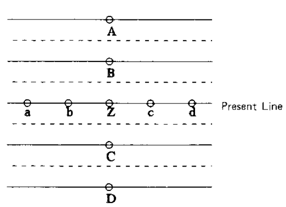


FIGURE 14 Pixel configuration used in the moving-area segmentor.

$$\mathrm{MAD}(j,k) = \frac{1}{M^2} \sum_{m=1}^{M} \sum_{n=1}^{M} |f_i(m,n) - f_{i-1}(m + j, n + k)| \tag{37}$$

Several searching algorithms have been proposed to reduce the number of steps to reach the matching point. The two-dimensional logarithmic search procedure is one of them and is proposed by J. R. Jain and A. K. Jain [*17*]. The procedure is well demonstrated in Figure 15. It is assumed that the matching criterion, MAD, increases monotonically as the search moves away from the right point. In each searching step, 5 points are checked (as shown in Fig. 15). The search step distance is reduced if the minimum MAD is in the center point or at the boundary of the searching area. Otherwise, the point with minimum MAD will be taken to be the new central searching point.

2.6.2 Coding of Color Images

A color image is usually represented by independent red, green, and blue images. If R G B images are coded separately, the coding error and channel error will affect visual quality of the reconstructed color image a great deal due to the characteristics of human color perception. Frei and Baxter investigated the model of human color perception and proposed a nonlinear transform based on this color perception model [*18*]. This color-perception-based nonlinear transform consists of three operations and is best illustrated by the block diagram in Figure 16.

The first two operations are mathematically defined as

$$\begin{bmatrix} T_1 \\ T_2 \\ T_3 \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.144 \\ 0.607 & 0.174 & 0.201 \\ 0.0 & 0.066 & 1.117 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{38}$$

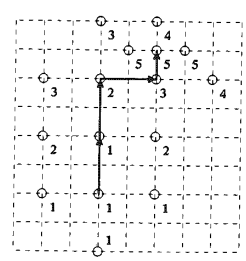


FIGURE 15 Illustration of two-dimensional block searching.

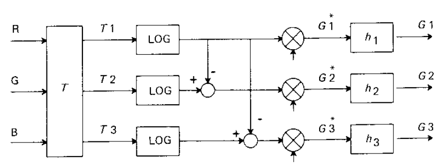


FIGURE 16 Block diagram of nonlinear transform based on the color perception model.

and

$$\begin{bmatrix} G_1^* \\ G_2^* \\ G_3^* \end{bmatrix} = \begin{bmatrix} 21.5 & 0 & 0 \\ -41.0 & 41.0 & 0 \\ -6.27 & 0 & 6.27 \end{bmatrix} \begin{bmatrix} \log T_1 \\ \log T_2 \\ \log T_3 \end{bmatrix} \tag{39}$$

Three band-pass filters (the third operation) are plotted in Figure 17. The simulation results showed that the distortion computed with this model coincides well with subjective measures produced by a group of observers. With the same random noise added to R G B images and transformed G_1, G_2, and G_3, color image reconstructed from G_1, G_2, and G_3 appears to have smaller chromatic error, and the errors are distributed more uniformly with respect to spatial frequency. Therefore, its distortion is much lower than the color image reconstructed from RG B images.

Pseudocolor Coding: An alternative coding method for color images is based on pattern recognition techniques. Colors of an image represented by R G B primary color images can be plotted as points in R G B color space. For a particular image, either analogous or digital, its colors in color space usually appear as clusters. If a cluster is represented by one color, say, the centroid of the cluster, and all representative colors are labeled with color codes, a color image can then be coded by an image color code and a color codebook, which contains the color definition of

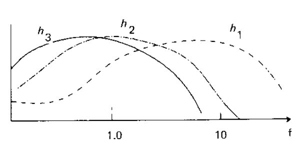


FIGURE 17 Band-pass filter functions of the color perception model.

each color code, the coordinates of each representative color in color space. To display the coded images, one only needs to load the color code image into a display buffer and the color codebook into a color look-up table. As a matter of fact, the principle of this color-coding scheme has been used in color textile printing and color publication printing for quite a long time.

An example of such a color-coding scheme was proposed by Wu [*18*]. Its block diagram is shown in Figure 18. Each pixel in its original color image is viewed as a pattern vector, $[f_r, f_g, f_b]$. To derive the knowledge of color clusters, the distribution of original color image in R G B space is first computed and then converted into a uniform color space. An unsupervised learning algorithm is performed to find color clusters, and their distribution parameters, namely means and populations. With the defined clusters, original image data are classified pixel by pixel, using a nearest neighbor classifier with consideration of cluster population to get the final output. Using the color-coding scheme introduced above, Wu was able to code color images using about 16 colors without noticeable distortion [*8*].

3. BINARY IMAGE CODING

Examples of binary images include documents, checks, weather maps, engineering drawings, geographic maps, newspaper pages, and so forth. A thorough review of coding techniques of binary images can be found in Ref. *20*. In this section, we shall review two main coding techniques for binary images: white block skipping (WBS) coding and run-length coding.

3.1 WBS Coding

Henceforth, suppose that the background of a binary image is coded as 0 or white, and an image object as 1 or black. In most cases, there are more white pixels than black in a binary image. Intuitively, skipping white pixels will significantly reduce the bit rate. Skipping white pixels can be acheived both in one dimension and two dimensions.

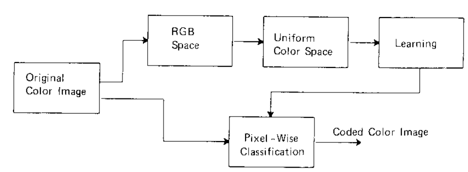


FIGURE 18 Block diagram of pseudocolor coding scheme.

The one-dimensional WBS coding method simply divides image lines into N pixel segments. A 1-bit code word of 0 is used for a white segment containing only white pixels. A segment with at least one black pixel is coded by an (N + 1)-bit code word, with the first bit being 1 and the remaining N bits being the same as the original data. For example, if the original data are 00001011, and the segment size N = 4 is chosen, the one-dimensional WBS code of the data is 011011.

The bit rate of one-dimensional WBS coding method is

$$\text{bpp} = (1 - p_w + \frac{1}{N}) \quad \text{bit/pixel} \tag{40}$$

where p_w is the probability of the white segment. For a given set of images, p_w can be measured experimentally. N is the length of the segment. Obviously, N depends on the resolution of images. Many experimental results have shown that a value of N as approximately 10 is quite suitable for a wide range of images.

Two-dimensional WBS coding is exactly an extension of one-dimensional WBS coding. Instead of a one-dimensional segment, image block size MxM is used. Again we use a 1-bit code word 0 for a white block, and a $(M^2 + 1)$-bit code word for blocks containing at least one black pixel. Because of the two-dimensional correlation property of images, two-dimensional WBS coding is naturally superior to one-dimensional WBS.

The WBS coding method introduced above has a fixed structure despite the local texture changes. A more efficient adaptive WBS coding method using hierarchical block size was proposed by DeCoulon and Johnsen [*21*]. Suppose the coding method starts from a block size MxM, with M = 16; if the whole block is white, a 1-bit code word 0 is assigned. Otherwise, we prefix with a 1 and divide the block into four 8x8 subblocks. For each subblock, we repeat the same coding process until the block size is reduced to 2x2.

3.2 Run-Length Coding

As indicated by the name, run-length coding codes the gray-level run lengths rather than the gray level itself. For binary images, the gray-level information is implicity included in run length; we need only to declare the gray level at the beginning of a line if necessary.

> *Example*: A run-length code of 24,2,68,2,33 represents a binary image line; the line length is 128 pixels. After 24 white pixels there are 2 black pixels, followed by 68 white, 1 black, and 33 white pixels.

The dynamic range of the run-length number can be quite large. For an image of 1024 × 1024, the run length can take the value of 1–1024. Therefore, it is not an efficient way to code run lengths with code words of fixed length. Let (1,2,...,N) be the set of values the run length takes, where N is the image line length. A set of probabilities $(p_1, p_2, \ldots, p_N)$ is associated with the run-length value set and can be measured experimentally over a set of images. Based on the set of probabilities, a set of variable-length codes can be designed, which uses short length codes for most frequently occurring numbers to achieve efficiency. Hoffman code is

one of most efficient codes of variable length, but it is complicated to implement. In the following sections, we describe two classes of suboptimum codes: linear codes and logarithmic codes, which are particularly suitable for binary image coding.

3.2.1 Linear Codes

For description convenience, let us define a message set (m_k), a probability set (p_k), and a set (b_k), $k = 1,2\ldots,N$, where p_k is a probability associated with message m_k and b_k is the number of bits in the code word for m_k. b_k of linear codes is proportional to k. Laemmel code, from the class of linear codes, is suitable for run-length coding of line drawings and is described by the following example:

Laemmel code is usually denoted by L_N, where N is the code block size. Let

$$k = q(2^N - 1) + r \tag{41}$$

where

$$q = 0,1,2,\ldots \quad \text{and} \quad 1 \leqslant r \leqslant 2^N - 1$$

The code word for m_k consists of (q + 1)N bits. The first qN bits are 0's, and the last N bits are binary representations of r. If we use L_3 code to code message m_{15}, as $15 = 2(2^3 - 1) + 1$, the code is 000000001. Similarly, L_4 code for m_{15} is 00000000. Obviously, there is an optimum N for a class of binary images. It is the integer nearest to $(1 + \log_2 a)$, where a is the mean of the geometrical distribution.

3.2.2 Logarithmic Codes

The code word length b_k of logarithmic codes is approximately proportional to the logarithm of k. Among logarithmic codes, Hasler codes (H_N codes) are most suitable for run-length coding of text binary images. The H_N codes are constructed as follows. The all-possible N-bit words, 2N-bit words, 3N-bit words, and so forth, are first listed; an additional bit 1 is inserted on the end of the code word and an additional bit 0 is inserted between any two N-bit blocks. Following is the H_1 code; the inserted bits are underlined.

$0\ \underline{1}$
$1\ \underline{1}$
$0\ \underline{0}\ 0\ \underline{1}$
$0\ \underline{0}\ 1\ \underline{1}$
$1\ \underline{0}\ 0\ \underline{1}$
$1\ \underline{0}\ 1\ \underline{1}$
$0\ \underline{0}\ 0\ \underline{0}\ 0\ \underline{1}$
$0\ \underline{0}\ 0\ \underline{0}\ 1\ \underline{1}$

Huang showed that adaptive WBS coding and run-length coding of a set of binary images have more or less similar data compression ratio [*20*].

4. OBJECT-ORIENTED IMAGE CODING

The image-coding techniques we have discussed so far are mainly image oriented or texture oriented. Efforts have been made to develop efficient methods that can preserve image texture with the least bits per pixel as possible. Most efficient techniques are texture adaptive.

This section is devoted to object-oriented image coding, which is mostly used in pictorial information systems, CAD systems, and computer vision systems for the purpose of object management. The functions of object-oriented image coding include storage, retrieval, display, plotting, and reasoning. We are most likely involved in binary images. They are either segmentation results of gray-level images or line drawings and geographic maps. Their spatial relationships and the semantic interpretations of objects are of interest to users. Let us consider, for example, a forest geographic information system and define a query: Plot an area map where the pine grows and the elevation of the area is between 500 and 800 meters above sea level. To process this query, one may first find the area where the pine grows and the area between 500 and 800 meters and then perform the set operation "intersection." The second example can be a robotic vision system. A command "go and pick up a stool on the left of the chair" is given. After receiving the command, the robot may first find the right position of the chair and then look for the stool on the left of the chair. We note from these two examples that the object-oriented image-coding techniques should be carefully designed in order to meet the requirements of object management tasks.

All objects contained in binary images fall into three categories: point, line (straight and curve), and region. Therefore, the spatial relationships between objects, can be summarized as

1. Close to (in distance). This spatial relationship is defined by the distance between objects. The definition is rather fuzzy. Other relations, to the left (right) of, to the east (west, north, south) of, may belong to this category as well.
2. Contained in. A point, a line segment, or a small region can be contained in a large region.
3. Intersection. It exists among lines and regions.
4. Surrounded by or partially surrounded by.
5. Pass through. A line object passes through a region.

Set operations are usually invoked to check the spatial relationships of objects physically coded by the coding methods to be described in this section.

Now we can summarize the requirements of object-oriented image coding with applications to pictorial information systems, CAD systems, and computer vision.

1. Efficient to code all three types of objects: point, line, and region.
2. Easy for neighbor finding operations.
3. Easy for set operations.
4. Easy to compute geometric properties.
5. Easy for geometric transforms, such as shift, scale, variation, and rotation. Scale variation is important for systems requiring zooming.

6. Compatible with raster data structure and vector data structure. Image display requires raster data structure, and plotter uses vector structure. These two types of devices are used most frequently in pictorial information systems.
7. Compact and easy to access.

In the following sections, three types of object-oriented coding techniques are described: chain coding, run-length coding, and quadtree and related hierarchial data structures. Among them, chain coding is basically line-pattern coding; the later two are basically region coding. Each has its own advantages and disadvantages and is applicable in certain situations.

4.1 Chain Coding

4.1.1 Coding Scheme

Because the information of a region is actually contained in its boundary, the line pattern coding technique, chain coding, is capable of coding three types of image objects. The idea of chain coding is to follow line or boundary points and to code them in sequence. A well-known chain-coding method is described extensively by Freeman [*22*]. The chain-coding scheme can be stated as

1. A link, α_i, is a directed straight-line segment of length $(\sqrt{2})^p$ and of angle $\alpha_i \times 45^0$ referenced to the x-axis of a right-hand Cartesian coordinate system, where $\alpha_i = 0,1,\ldots,7$, $p = \mathrm{mod}\,(2, \alpha_i)$.
2. A chain is an ordered sequence of links with possible interspersed signal codes.

$$A = a_1\ a_2\ \cdots\ a_n$$

A signal code is usually taken as an octal digit sequence $04d_1d_2$; for example, 0400 denotes the end of the chain, 0407uv denotes the serial number uv of the chain, 0424xyz the gray level, and 0426xyz and 0427xyz denote absolute x- y-coordinates. For a closed curve in Figure 19, the chain code would be

0407010424001042600104270031007056423552204 00

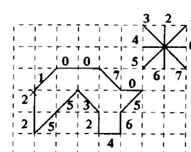


FIGURE 19 A chain-encoded closed contour.

4.1.2 Operations on Chain-Coded Objects

1. Inverse of a chain
   The inverse of a chain is a geometric congruent of the chain and is oppositely directed.

$$(a_1 \cdots a_n)^{-1} = a_1^{-1} \cdots a_n^{-1} \tag{42}$$

   where $a_i^{-1} = \mathrm{mod}[8, (a_i + 4)]$
2. The length of a chain can be easily computed as

$$L = T(n_e + n_o \sqrt{2}) \tag{43}$$

   where n_e and n_o are the numbers of even- and odd-valued links, respectively. The absolute coordinates of a point, shown by link a_i, are

$$x_i = \sum_{j=1}^{i} a_{jx} + x_0 \tag{44-1}$$

$$y_i = \sum_{j=1}^{i} a_{jy} + y_0 \tag{44-2}$$

   where a_{jx} and a_{jy} are x and y components of a_j, respectively.
3. The width and the height of an enclosed contour is

$$\text{width} = \max_j x_j - \min_j x_j \tag{45-1}$$

$$\text{height} = \max_j y_j - \min_j y_j \tag{45-2}$$

4. The distance between two points connected by a chain is

$$d = \left[\left(\sum_{i=1}^{n} a_{ix}\right)^2 + \left(\sum_{i=1}^{n} a_{iy}\right)^2\right]^{1/2} \tag{46}$$

By knowing the coordinates of each point on the contour, the moments of the contour can be computed. But difficulty in calculating the geometric properties of the region enclosed by a chain-coded contour from the chain code and the scale variation can also present a problem.

Alternatives for chain coding may be directly coding the line pattern by sequentially collecting the coordinates of points along lines or the coordinates of vertices of polygons and piecewise linear curves.

4.2 Object-Oriented Run-Length Coding

4.2.1 Coding Scheme

Different from the run-length coding technique (Section 3.2), object-oriented run-length coding can be considered as a variation of chain coding. To display a chain-coded image region on a raster-type display screen, the boundary points should be first sorted according to the y-axis coordinate, and the points with the same y-axis are sorted according to the x-axis coordinate. By parity checking, the region can then be filled line by line in a raster data structure manner.

$$y_1, x_{11}, x_{12}, \cdots$$
$$y_2, x_{21}, x_{22}, \cdots$$
$$\vdots$$

To facilitate the manipulation of object regions, object-oriented run-length coding codes the region objects using the following form. It is illustrated in Figure 20.

$$\{y_1\,(x_{11}, l_{11}, x_{12}, l_{12}, \cdots),\ y_2\,(x_{21}, l_{21}, x_{22}, l_{22}, \cdots),\ \cdots\}$$

To form a complete relational description of a region object, an object name and several object attributes can be inserted in front of this data record. Probably, the maximum and minimum of x- and y-coordinates can give a quick estimation of object position. Other geometric properties, such as area, long axis, short axis, and elongation, may also be useful in some applications. One may then have

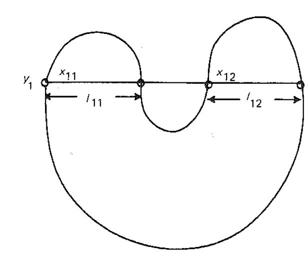


FIGURE 20 Illustration of object-oriented run-length coding.

$$\{\text{ob jname, centroid, } x_{max}, x_{min}, y_{max}, y_{min}, L_{axis}, S_{axis}, \\ y_1(x_{11}, l_{11}, x_{12}, l_{12}, \cdots), \cdots\}$$

The same coding scheme is applicable to point objects and line objects. In the case of a point object, the data record is simplified as $[y(x,1)]$.

4.2.1 Operations on Run-Length-Coded Objects

Let two objects be represented by

$$\{O_1, C_1, x^1_{max}, x^1_{min}, y^1_{max}, y^1_{min}, y^1_1(x^1_{11}, l^1_{11}, \cdots), \cdots\}$$
$$\{O_1, C_1, x^2_{max}, x^2_{min}, y^2_{max}, y^2_{min}, y^2_1(x^2_{11}, l^2_{11}, \cdots), \cdots\}$$

The union of two objects O_1 and O_2, then consists of the following steps:

1. If $x^1_{max} < x^2_{min}$ or $y^2_{max} < x^1_{min}$ or $y^1_{max} < y^2_{min}$ or $y^2_{max} < y^1_{max}$, the two objects are separated from each other, the run-length code of the union of the two objects is obtained simply by combining two run-length codes and by updating the object name and the attribute values.
2. If not the case of 1, do the following:
   a. For each line $y_i{}^1$, if $y_i{}^1 < y^2_{min}$ or $y_i{}^1 > y^2_{max}$, put the data of line $y_i{}^1$ into the result record. Otherwise, find the corresponding line y^2_m in O_2. For each run $x_i{}^1{}_j$, $l_i{}^1{}_j$, Check if there is any (can be more than one) run x^2_{mn} , l^2_{mn}, such that $x_i{}^1{}_j < x^2_{mn} \leqslant x_i{}^1{}_j + l_i{}^1{}_j$ or $x^2_{mn} \leqslant x_i{}^1{}_j \leqslant x^2_{mn} + l^2_{mn}$. If not, take run $x_i{}^1{}_j$, $l_i{}^1{}_j$ from O_1 and put it into result record. Otherwise, combine run $x_i{}^1{}_j$, $l_i{}^1{}_j$ with all runs in O_2 that satisfy the above conditions, put the combination into the result record, and remove all involved runs from their original places. The combination of run $x_i{}^1{}_j$, $l_i{}^1{}_j$ and run x^2_{mn}, l^2_{mn}, x_r, l_r is

   $$x_r = \min(x^1_{ij}, x^2_{mn}), \quad l_r = \max[(x^1_{ij} + l^1_{ij}), (x^2_{mn} + l^2_{mn})] - x_r$$

   Put any runs left in y^2_m into the result record. For this line in result data record, check if any runs overlap. Combine overlapped runs until there are none.
   b. Put the data left in O_2 into the result record.

3. Assign a new object name and compute attribute values for the result.

Intersection operation of two objects is more simple than union.

1. If $x^1_{max} < x^2_{min}$ or $y^2_{max} < x^1_{min}$ or $y^1_{max} < y^2_{min}$ or $y^2_{max} < y^1_{max}$, the two objects are not intersected.
2. For each line $y_i{}^1$ in O_1, check if there is a line $y_m{}^2$ such that $y_i{}^1 = y_m{}^2$. If $y_m{}^2$ does not exist, do nothing. Otherwise,
   a. For each run $x_i{}^1{}_j$, $l_i{}^1{}_j$, if there is a run x^2_{mn}, l^2_{mn} such that $x_i{}^1{}_j < x^2_{mn} \leqslant x_i{}^1{}_j + l_i{}^1{}_j$, put the run x^2_{mn}, $\min[(x_i{}^1{}_j + l_i{}^1{}_j), (x^2_{mn} + l^2_{mn})] - x^2_{mn}$ into the result record; remove these two runs from their places. If $x^2_{mn} \leqslant x_i{}^1{}_j \leqslant x^2_{mn} + l^2_{mn}$, the intersection of these two runs is $x_i{}^1{}_j$, $\min[(x_i{}^1{}_j + l_i{}^1{}_j), (x^2_{mn} + l^2_{mn})] - x_i{}^1{}_j$.

3. Assign an object name and compute the attribute values for result data.

The procedure to check containment of two objects is the same as the intersection operation, except that each run of an object should be contained totally within that of the other object.

Computing the geometric properties of objects encoded by run-length coding is a trivial process because it involves only point operations that are fully supported by run-length coding. For example, to compute the area of an object region, one needs only to sum up all run lengths.

$$\text{area} = \sum_i \sum_j l_{ij}$$

4.3 Quadtree

Quadtree is a hierarchial data structure providing a quick data access in the sense of data retrieval. An extensive review of quadtree and related hierarchial data structure can be found in Ref. *23*.

Quadtree is based on the principle of recursive decomposition of space. Assume that we are only interested in digital images in a two-dimensional grid. Each decomposition produces four equal-sized quadrants. Label the quadrant white if it consists of only white pixels, black if it consists of only black pixels, and gray if it consists of both black and white pixels. Further decomposition is carried out on gray blocks; black and white blocks remain unchanged. The decomposition process is best described by a quadtree with its root representing the entire image and the leaf representing either black or white quandrants. Figure 21 is an example of coding an object region by quadtree data structure. In the binary image (a), the background is coded with 0's and the object by 1's. Blocks within the object region are shaded in b, the decomposition of the image in a; A quadtree representation of b is shown in c. We notice, from Figure 21, b, that in quadtree representation, object region is now considered as composed of large square blocks instead of pixels. This is the reason why quadtree can represent image data in a more compact way than raster data structure.

4.3.1 Ways of Representing Quadtrees

A natural way of representing quadtree is to use a tree structure, as shown in Figure 21, c. Each node of the tree is represented by a record

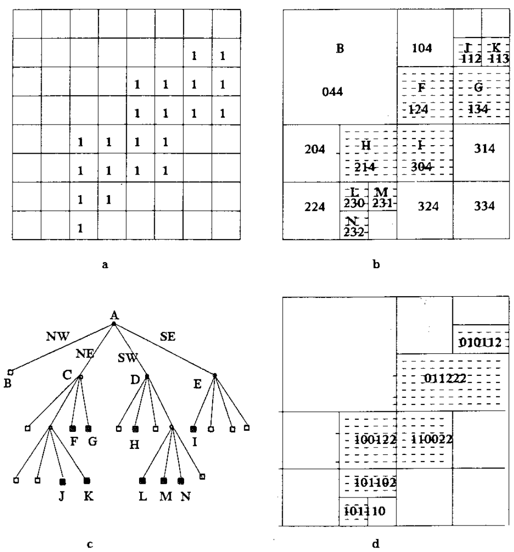


FIGURE 21 (a) An object in a binary image; (b) quadtree decomposition of the image in a; (c) quadtree representation of a; (d) binary tree decomposition of image in a.

with four pointers to its four son records and one pointer to its father record. The use of a pointer facilitates tree traversal. Operations based on tree traversal can be performed without any difficulties.

The number of nodes of quadtree, when representing an image, may be quite large, sometimes even larger than the number of pixels of the image. Therefore, representing a quadtree using tree structure has the disadvantage of requiring too much storage. It is described in a number of publications, [e.g., *24*]. Location code is a base 4 number, with code 0,1,2,3 corresponding to, for example, quadrants NW, NE, SW, and SE, respectively. Code 4 denotes a "don't care," indicating no decomposition in this level. Assuming that an image sixe is $2^N \times 2^N$, the location code will be N digits long. A leaf corresponding to a $2^k \times 2^k$ block will then be coded by a location code having $(N - k)$ don't care digits.

An object region can then be represented by a collection of location codes of image blocks of which the object region consists. For example, the object region in Figure 21 is coded by linear quadtree as

{112,113,124,134,214,230,231,232,304}

The location codes listed correspond to blocks J,K,F,G,H,L,M,N, and I, respectively. It is interesting to note that the location list above is in increment order, and the corresponding block list is a depth-first traversal of black nodes of the quadtree.

Kawaguchi et al. [*25*] proposed a quadtree representation using the depth-first traversal list of a quadtree leaf. The list is a string consisting of symbols ("(","B,""W"), corresponding to gray, black, and white label. The quadtree in Figure 21 is coded by Kawaguchi's method as

(W(W(WWBBBB(WBW(BBBW(BWWW

The original image is reconstructed from depth-first (DF) expression by the fact that the degree of each terminal node is always 4 and the traversal is in the order of NW, NE, SW, SE.

Among three representations introduced above, linear quadtree is a most abstractive one. It is compact, and the operations based on it can be defined as algebraic algorithms. In the following sections, we will describe properties of linear quadtree and the operations on objects represented by linear quadtree.

4.3.2 Properties of Linear Quadtree

Suppose a block is coded by a location code

$$a_1 a_2 \cdots a_i \cdots a_N$$

where $a_1, a_2, \cdots, a_i \neq 4, a_{i+1}, \cdots, a_N = 4$. Then the size of the block is $2^{N-i} \times 2^{N-I}$; the coordinates in the upper left-hand corner of the block are

$$x = \sum_{j=1}^{i} \mathrm{mod}(2, a_j) 2^{N-j} + 1 \qquad (47\text{-}1)$$

$$y = \sum_{j=1}^{i} [a_j - \mathrm{mod}(2, a_j)] \; 2^{N-j-1} + 1 \qquad (47\text{-}2)$$

We now try to develop rules to determine whether two given blocks are neighbors. It is worth noting that the concept neighbor here refers to direct neighbors side by side. Suppose the location codes of two blocks are

$$a_1 a_2 \cdots a_i \cdots a_N$$

$$b_1 b_2 \cdots b_j \cdots b_N$$

Again $a_{i+1}, \cdots, a_N$ are supposed to be 4's as are $b_{j+1}, \cdots, b_N$. Without loss of generality, we can assume $i \leqslant j$; this means block A is greater than or equal to block B. The neighbor identification operation is then as follows:

1. Compare a_k with b_k for $1 \leqslant k \leqslant i$. If $a_k - b_k = m$, $|m| < 3$, both ($a_k = 1, b_k = 2$) and $a_k = 2, b_k = 1$) are not true. (The last two conditions are to guarantee direct neighbor relation; in 8-neighbor definition, blocks corner by corner are also neighbors, and these two conditions can be neglected.) Block A and block B are neighbors is $a_n - b_n = -m$ for k n i and
   a. If m = 1, block B is to the west of block A. $b_{i+1}, \cdots, b_j$ should be either 1 or 3. For example, block 324 and 233 in Figure 21 are neighbors.
   b. If m = -1, block B is to the east of block A. $b_{i+1}, \cdots, b_j$ should be either 0 or 2. For example, block 044 and 104, 124 are neighbor blocks.
   c. If m = 2, block B is to the north of block A. $b_{i+1}, \cdots, b_j$ chould be either 3 or 2.
   d. If m = -2, block B is to the south of block A. $b_{i+1}, \cdots, b_j$ should be either 0 or 1. For example, 044 and 204, 214 are neighbors.

4.3.3 Operations on Objects Represented by Linear Quadtree

Operations on linear quadtrees can be performed by algebraic algorithms. We describe here mainly the set operations and connected component labeling.

Set operations on two object regions are based on containment checking between two image blocks. Let us use the same expression as the previous subsection for blocks A and B. Block A is contained in (or equal to) block B, if and only if $i > j(i{=}j)$ and $a_k = b_k$ for all $0 \leqslant k \leqslant j$.

Union operation on two object regions $A = \{A_1, A_2, \cdots\}$ and $B = \{B_1, B_2, \cdots\}$ are performed by checking containment between blocks within the two object regions.

1. From each block A_i in object region A, check if there is a block B_j in object region B which contains or equals A_i.
   a. If this is true, place B_j into result region and remove B_j from B, A_i and other blocks contained in B_j from A.
   b. If it is false, place A_i into result, remove A_i from A and any blocks contained in A_i from B.
2. Place all blocks left in B into result.

Intersection operation of two object regions is principally the same as union operation, except that smaller blocks are placed into the result region instead of larger ones. It is worth noting that spatial relationships between two blocks of two object regions represented by linear quadtree are only "contain in," "equal to," and "close to" if the binary images have the same size. This property of the quadtree makes the set operations simple.

Labeling of connected components in an object region requires clear definition of connectedness on quadtree representation. Analogous to the connectedness definiton of regions in a square grid, we define A to be connected if for any two blocks A_1 and A_N in A, there is a sequence of N blocks $A_1, A_2, \cdots, A_N$ such that

$$A_i \cup A_{j+1} = 0 \qquad 1 \leqslant i < A_{N-1}$$

A_i is a direct neighbor of A_{i+1}

The labeling of connected components in an object region can be completed by the following algorithm:

1. Take a block from block list A of a region as an initial block of a component and label it L.
2. Scan the rest of the blocks in list A. For each block in A, check if it is a neighbor to any blocks already in a component list C. It it is, put the block into C.
3. Repeat step 2 until no blocks can be put into C.
4. Increment L; go back to step 1.
5. Terminate the algorithm if list A is empty.

Computing the area of a region represented by linear quadtree is a trivial process. The area of each block is simply $2^n \times 2^n$, where n is the number of 4's in the location code. The areas of blocks are then summed up to get the overall area of a region.

The moments are also computed block by block. For each block, the coordinates of each pixel can be easily calculated by Eq. (47); computing moments is then a straightforward process.

4.3.4 Binary Tree Representation

An alternative hierarchial data structure is the binary tree [26]. The recursive decomposition of the binary tree divides images into two equivalent parts by cuting along the x-axis and then along the y-axis, resulting in "left" and "right", and "above" and "below" parts, respectively. Figure 21, b shows a binary tree decomposition of images in a. As can be seen from Figure 21, b and d, the binary tree representation requires less (or equal) leaves to represent an object region. This is because its leaves correspond either to squares or rectangles.

If we label "left" and "above" by 0, and "right" and "below" by 1, a binary tree representation similar to linear quadtree is obtained. The operations on binary-tree-represented objects can be derived analogous to linear quadtree.

4.4 Conversions between Object-Oriented Image Coding

It is convient to do conversions by relating all object-oriented image codings to raster data structure. Raster data structure is a natural data

structure of images. Conversions between raster data structure and object-oriented representations are well specified by their definitions.

5. CONCLUSIONS AND REMARKS

In this article we have discussed three groups of image data compression techniques. Gray-level image data compression is not error free. Research efforts have been made to develop techniques to reduce the data rate while keeping the error rate as low as possible. Binary image coding is error free. That is, no coding error is permitted. Object-oriented image coding is mainly for image data management; Therefore, low data rate is not the main goal. Instead, performance requirements listed in Section 4 are important.

Advances in image processing, pattern recognition, pictorial information systems, and other related research fields are adding more and more approaches to image coding. Many interesting advances can be anticipated.

REFERENCES

1. M. D. McFarlane, "Digital Picture Fifty Years Ago," *PIEE 60* (7) (1972).
2. E. J. Delp and O. R. Mitchell, "Image Compression using Block Truncation Coding," *IEE Trans. Commun., COM-27*(9), 1335–1342 (1979).
3. E. J. Delp, R. L. Kashyap, and O. R. Mitchell, "Image Data Compression using Autoregressive Time Series Models," *Pattern Recognition, 11*, 313–323 (1979).
4. T. Max, "Quantizing for Minimum Distortion," *IRE Trans. Inf. Theory, IT-16*, 7–12 (1960).
5. R. Schafer, "DPCM Coding of the Chrominance Signals for the Transmission of Color TV Signals at 24 Mbits/s," *Signal Processing*, (1981).
6. C. Zhang, "Ein Neuer Adaptiver Pradiktor fur die DPCM-Codierung von Fernsehsignalen," *Frequenz, 36*, 161–184 (1982).
7. J. K. Wu and R. E. Burge, "Adaptive Bit Allocation for Image Compression," *CGIP 19*, 392–400 (1982)
8. J. K. Wu, "Color Image Coding of Images," *Signal Processing* (October 1986).
9. D. J. Healy and O. R. Mitchell, "Digital Video Bandwidth Compression using Block Truncation Coding," *IEEE Trans. Commun., COM-29* (12), 1809–1817 (1981).
10. M. Kocher and M. Kunt, "A Contour-Texture Approach to Picture Coding," *Proc. ICASSP-82*, Paris, France, May 1982, pp. 436–440.
11. A. Rosenfield and K. Kak, *Digital Picture Processing*, 2nd ed., Academic Press, New York, 1983.
12. C. P. Wang, "Image Data Compression using a Codebook", Masters dissertation, Illinois Institute of Technology, Chicago, IL, December 1985.
13. B. Ramamurthi and A. Gersho, "Image Vector Quantization with a Perpetually Based Cell Classifier," *IEEE Int. Conf. ASSP*, 32.10.1–32.10.4 (1984).

14. T. S. Huang, *Image Sequence Processing and Dynamic Scene Analysis*, Springer-Verlag, 1983.
15. A. N. Netravali and J. D. Robbins, "Motion Compensated Television Coding–Part I," *Bell Syst. Tech. J., 58*, 631–670 (March 1979).
16. C. Cafforie and F. Rossa, "The Differential Method for Image Motion Estimation, " in *Image Sequence Processing and Dynamic Scene Analysis* (T. S. Huang, ed.), Springer-Verlag, 1983, pp. 104–124.
17. J. R. Jain and A. K. Jain," Displacement Measurement and Its Application in Interframe Image Coding," *IEEE Trans. Commun., COM-29*, 1799–1806 (1981).
18. W. Frei and B. Baxter, "Rate-Distortion Coding Simulation for Color Images," *IEEE Trans. Commun., COM-25*(11), 1385–1392 (1977).
19. J. K. Wu and W. M. Zhang, "Adaptive Transform Image Compression by using Texture Analysis, *Commun. China*, (1987, in press).
20. T. S. Huang, "Coding Two-Tone Images," *IEEE Trans. Commun., COM-25*(11), 1406–1424 (1977).
21. F. DeCoulon and O. Johnsen, "Adaptive Block Scheme for Source Coding of Black and White Facsimile," *Electron. Lett., 12*(3), 61–62 (1976).
22. H. Freemen," Computer Processing of Line-Drawing Images," *ACM Comput. Surv., 6*(1), 57–97 (march 1974).
23. H. Samet, "The Quadtree and Related Hierarchial Data Structures," *Compu. Surv., 16*(2), 187–260 (1984).
24. D. J. Abel and J. L. Smith, "A Data Structure and Algorithm Based on a Linear Key for a Rectangle Retrieval Problem," *CVGIP, 24*(1), 1–13 (1983).
25. E. Kawaguchi, T. Endo, and M. Yokota, "DF-Expression of Binary-Valued Picture and Its Relation to Other Pyramidal Representations," *Proceedings of the Fifth International Conference on Pyramidal Representations*, December 1980, pp. 822–827.
26. Y. Ohsawa and M. Sakauchi, "*The BD-Tree—A New N-Dimensional Data Structure with Highly Efficient Dynamic Characteristics,*" in *Information Processing*(R. E. A. Mason, ed.), Elsevier Science Publishers B. V. North-Holland, Amsterdam, 1983, pp. 539–544.

SHI-KUO CHANG
JIAN-KANG WU

COMMAND LANGUAGES

A command language is a language utilized to issue commands to a particular piece of software or hardware. Examples are the languages utilized to give commands to an editor, an operating system, or the computer hardware itself (machine language). The essence of a command language can be understood by considering the command language interpreter. The central feature of a command language interpreter is the read, interpret, execute cycle. The interpreter reads the command (from the input line or, in the case of the machine, from a memory location), interprets (decodes) it, and carries it out. As an example, consider the activity of an operating system.

The user of the operating system enters commands at the terminal or other device, for example, the command to print the contents of a file:

```
type myfile
```

The operating system reads each command entered, determines which command it is (type) and the arguments to the command, if any (myfile), and executes it. The essence of the process is the cycle

```
Loop
  -read
  -interpret
  -execute
```

The loop is an "infinite" loop, terminated by a command to log off (operating system), quit or end (editor), or other means of indicating that a session is over.

The distinction between the languages referred to as programming languages and those referred to as command languages is that between the accept/execute mode of the command interpreters and the specification of a sequence of commands to be carried out in some particular order, usually governed by the input as well as the program, at a later time. As the mention of machine language as a command language and the machine as the interpreter indicates, what one considers to be a command language depends on the level of analysis. At the level of computer hardware, the command interpreter is the instruction decoder, which accepts instructions fetched under the control of the program counter or instruction counter, decodes them through the circuitry, and causes the hardware to execute them. The entire session is ended with a command to halt, which usually triggers an interrupt to the operating system, the commands of which are executed on the same piece of hardware [*1*]. In the recent literature, however, the term command language has been used primarily to refer to languages in which an individual interacts with a machine in real-time mode to accomplish some task [*2–5*]. This is usually taken to refer to languages used to interact with editors, operating systems, or application programs [*6–9*].

Some examples of command languages are the languages used to interact with the operating systems MS-DOS or UNIX, the editors available with the personal computer (PC) or UNIX-based systems, or the applications programs used by a teller in a bank or an airline reservation clerk. Because it is convenient and readily understood, the first example discussed is that of a line editor analagous to "ed," an editor utilized with UNIX or UNIX-like operating systems. Similar line editors are available with other micro-based operating systems. (A line editor sees the text as a series of lines or records, which are "gone to" to be manipulated; a screen editor sees the text as a two-dimensional screen of characters that can be accessed by the use of the cursor. The latter is often more convenient to use, but the former is explained more readily in print.)

Typical commands to a line editor are those to insert, delete, or change text, move the line pointer to a particular line in the buffer (area holding the text to be edited), or print a line of text or a set of lines. With ed, these tasks are accomplished by commands such as

3 — move to line 3
a — append the text that will be typed to the current line, that is, at the next line position
p — print the current line
d — delete the current line
s/pattern/new-value/—find the text in the pattern (in the current line) and substitute the new value for it

OPERATION OF THE COMMAND INTERPRETER

The operation of the command interpreter is to read the command,

p 3 (print line 3)

determine which of the commands it is

3|a|p|d|s (the vertical line, alternation, or "or", indicates a choice, that is, 3 or "a" or ...)

and any arguments to the command, and execute that command. One means of doing this is to keep the list of legal commands in a table, along with information indicating what routine (module) is to be executed if that command is encountered. The command is read from the input device and matched against the table entries, and if a match occurs, the routine indicated in the table is executed (branched to). If no match occurs, the command is illegal, and an error message is printed. This type of organization is often referred to as a "jump table" or a "dope vector." The phrase jump table comes from the use of a program module that employs a jump statement to branch to the routine listed in the table. The analogy in higher-level languages would be the use of a case statement, computed "goto" or other multiway selection statement. The phrase dope vector comes from the form of the table—the list of commands and addresses of the routines that execute them—which gives the information, or "dope," about each.

As a concrete example, consider the subset of commands

```
3
p
d
```

The task of the command interpreter is to accept the input string from the terminal, decide which of the three commands it is, and branch to that routine. Using a higher-level language, this would be accomplished with a read statement, a case statement or its equivalent, and a set of procedures. The structure of the code would be something of the form

```
read (command)
while (command < > 'exit') do
    case command of
        ['0'–'9']: process-move
               p: process-print
               d: process-delete
               other: process-error
    end {case}
```

The procedures process-move, process-print, process-delete, and process-error would contain the code to (a) locate the line pointer to the appropriate line in the editor's buffer (area of text being manipulated in the main memory), causing input/output (I/O) to and from the disk if necessary; (b) type out the appropriate text, for example, the current line or some range of lines (again, possibly causing I/O to and from the disk/buffer); (c) mark the line or lines as deleted (or remove them from the text), and adjust the line numbers of the remaining text accordingly (effectively "moving them up" in the file); (d) print an appropriate error message. As the example indicates, the movement to a line handles numbers in some given range, for example, 0–9 or 0–4096. One can also search for text patterns

```
/chris
```

And some commands take a variable number of arguments:

s/this/that/g	(to search for every occurrence of "this" on a line, substituting "that" for it)
s/this/that/	(to search only for the first occurrence of "this," changing it to "that")

The routines to recognize a command (parse it) and execute it are, therefore more complex than the example indicates. However, by abstracting from the details of the parsing module, the essential structure of the read/interpret/execute cycle is more clearly delineated.

Nonstructured code, employing a series of tests, to accomplish the same task would be of the form


```
10 Read command$
   If command$ = "0" ..."9" Gosub 1000
   If command$ = "p" Gosub 2000
   If command$ = "d" Gosub 3000
   If command$ < >
          ("0" ... "9" or "p" or "d") Gosub 4000
   Goto 10
```

The routines at 1000, 2000, 3000, and 4000 would be analogous to those described above. An alternative to the series of tests is the table lookup. The commands and a code value for each, referred to as a "token," are arranged in a table:

Command	Value
0–9	1
p	2
d	3

Upon reading the command, the command interpreter scans the table and, if the command is legal, uses the code value, or token, to control the execution of the appropriate routine:

```
Gosub 1000, 2000, 3000, 4000, on value
```

The code for the table lookup would be of the form

```
for j |=| 1 to n-of-commands
   if input-command |=| table-command(J)
     then
         Gosub 1000, 2000, 3000, on value (J)
     else
         Gosub 4000 {error}
```

This form of processing the command is analogous to the use of vectors in operating system tables, where the commands (or their codes) are associated with the addresses of the routines to process the command:

dir	512
type	256
ed	4096

As the example indicates, the routines need not be stored in the table in the numerical order of their locations. They would more likely be stored in some order that optimizes performance, for example, the relative frequency with which the commands are encountered. Also, the commands interpreted by the operating system may be commands to execute operating system routines (dir, type) or commands to execute other pieces of software (ed to transfer control to the editor; asm to request the assembler).

DESIGN OF A COMMAND LANGUAGE

One topic of concern in the current literature on human/machine communication is the design of command languages [*10–13*]. One issue addressed is the use of natural language (NL) versus the use of an artificially designed language. Another is the use of a certain syntactical format, such as prefix notation, versus another format, such as postfix. A third is the selection of the words (lexicon) to be used to name the commands (e.g., type vs. cat to print the contents of disk files). And a fourth is the use of function keys or a special I/O device such as a mouse, rather than keyed text to issue the commands.

NL

The issue of NL command languages versus artificial languages is more of a theoretical issue than one of practice. No current parser has the capability to interpret the full range of NL. However, several interesting attempts have been made to recognize NL in some restricted form or domain, and some of the techniques are applicable to the design of command language interpreters (the parser or portion of the system that decodes the command) and processors (the routines that carry out the command).

One of the earlier and more well-known attempts at human/machine dialogue was the Eliza project [*14*]. Eliza was a program that attempted to carry on a conversation with a human in a restricted domain of discourse. One of the early applications was in the domain of psychiatrist/client discourse. The common example is

Men are all alike.
  In what way?
They're always bugging me.
  Can you think of a specific example?
Well, my boyfriend made me come here.
  Your boyfriend made you come here.
He says I'm depressed much of the time.
  I am sorry to hear you are depressed.

The essence of the language interpreter/generator was a set of prestored patterns and responses to the patterns [*15*]. For example, given the pattern

$\underset{1}{\_\_\_\_}$ like(s) to play $\underset{2}{\_\_\_\_}$

where the blanks might be filled by names or pronouns ("John" or "I" in space 1) and various games or activities ("bridge" or "sports" in space 2), the response

Why do you like to play $\underset{2}{\_\_\_\_}$? (a)

or

Why does $\underset{1}{\_\_\_\_}$ like to play $\underset{2}{\_\_\_\_}$? (b)

might be generated. The choice of response a or b would be made on the basis of decision rules, such as the following:

If space 1 is a pronoun
(in this case I or we)
then print response a

If space 1 is a proper name
(of a person, pet, ...),
print response b

If neither of the above,
print response c:
  (c) Can you clarify that?

Systems such as this are referred to as rule-based systems in that they have a set of "if-then" rules that govern their behavior (responses). The if-then-else format is one that is suitable for a command interpreter, because it reflects the read/interpret/execute cycle:

-Read and interpret command
-If command |=|x then do y

Eliza was based on the storage of preset patterns of possible inputs and responses to these inputs. It scanned the input text for key words and printed an appropriate response to each. If it did not find a key word, it printed a noncommittal remark.

A similar methodology could be used in the design of command languages. The user of the system would enter the text in a relatively flexible (free) format:

Print line 3.
Would "you" please print line 3.
I'd like line 3 printed.

The parser would scan the commands for the key words or stems "print," "print-" and use these to trigger the appropriate routines. The arguments to the command module, for example, the range of line numbers, would be determined by the key words signaling the type of argument, "line(s)," and their values (the actual line number[s]). The remaining words would be treated as "empty" or "filler" words (sometimes called "noise" words, but the connotation is inappropriate in that they do not interfere with the message or, at least, are not intended to). Filler words are sometimes used with higher-level programming languages to allow the text to be more readable [*16*].

Hours worked Pic {is} 99.

The phrase in the brace is considered optional and will not affect the execution of the command.

If a key word is not encountered, an error message would be generated. Partial matches such as "prnt," as an approximation to "print," could be used to isolate keying errors, as well as to attempt to infer what command was intended [*17*]. More sophisticated techniques could also be used to infer the command that was intended [*18*].

Eliza was a system based on the storage of expected patterns and the responses to each. The pattern and key words were predefined for the intended domain of discourse. This format is suitable for the design of command languages, which usually concern a restricted domain of application, for example, editing files or interacting with an operating system, as well as a relatively restricted vocabulary and syntax for describing the tasks. Other early systems using restricted forms of syntax and restricted domains of discourse were the systems Deacon [*19*] and Baseball [*20*]. Deacon scanned input text for certain key words or symbols and used them, in conjunction with a dictionary, to interpret the text [*10*]. The text consisted of declarative sentences to be placed into a data base or queries of the data base. The declarative sentences were used to build the content and structure of the data base. Once the content was entered, the data base could be queried. The initial example was a military data base. One sample query was

Who is the commander of the 638th battalion?

The key phrase "Who is" indicates the place of the word or phrase to which a response is necessary; it follows the word "is." The system expects a set syntax:

Who is ______

The word "who" is a function word, indicating the position of the word or phrase to be responded to (whatever is in the blank or, in this case, the commander). The word "is" is filler. The word to be responded to is a content word and refers to one of the objects in the data base. The content words are designated R words (for "ring"). The structure of the data base is a system of interconnecting rings, or graph structure. That the text is a question is indicated by the question mark. (Declaratives end in an exclamation point.) The text is scanned to pick out the parts of the question, the words battalion, 638th, and commander, and these are interpreted with the use of the dictionary. The dictionary gives the parts of speech (roles) of the words and the connection between R (content) words and their representation in the data base, as well as their possible relationships to other words. The relative placement of the words

Who is $\underset{1}{\underline{\qquad}}$ (the commander) of $\underset{2}{\underline{\qquad}}$

in conjunction with the word "of," indicates that the item in space 1, commander, is an attribute of the word or phrase in space 2. The word or phrase in space 2 is decoded, in this case, into the noun, battalion, and modifier, 638th, which are looked up in the dictionary, along with the object of the query, the word commander. The dictionary, in conjunction with the data base structure (interconnected rings, or what would be called a semantic network in the current literature of computational linguistics), indicates that the word battalion does have an attribute, commander, associated with it. It also indicates that the modifier 638th is associated with the word battalion and that these two words, modifier and noun, can be grouped into one noun phrase, the 638th. The system is then able to reformulate the query as

Who is the commander of the 638th?

By continuing this process of reformulating the query and following the links in the data base, the system finally reduces the question to the form

Who is Jonathan M. Parker?

and returns that result, Jonathan M. Parker. Although the system is relatively complex for command languages associated with operating systems and editors, it is suitable for query language systems, that is, command languages used to interact with data bases, and has the advantage that the end user can design the data base (the objects and their relationships to one another, i.e., the network). (Function words indicate the structure of the text, i.e., the relationships among the words [their relative roles]; content words carry the "meaning" or semantic values of the words [or phrases]. Together, the two give the meaning of the sentence [*21*]).

Baseball is a system designed specifically to answer questions with respect to baseball data [*22*]. It has a data base containing the dates, places, teams, and scores of baseball games, stored in a fixed format (a set of lists). A typical entry is

Month |=| July
    Place |=| Boston
    Day |=| 7
    Game serial no. |=| 96
    (Team |=| Red Sox, score |=| 5)
    (Team |=| Yankees, score |=| 3)

Relationships, for example, the Yankees played the Red Sox, are indicated by including the items in the same list. The system responds to questions about the data base by analyzing the input to determine the parts of speech of the words (noun, verb), as well as their roles with respect to the domain of baseball (e.g., the word "score" indicates a query about the final result of the game). Typical queries are the following

Place |=| ?
Team |=| Red Sox
Month |=| July
Day |=| 7

can be generated to respond to the question, "Where did the Red Sox play on July 7"?. The query

Month |=| July
Place |=| Boston
Day |=| 7
Game serial no. |=| 96
(Team |=| Red Sox, score |=| ?)
(Team |=| Yankees, score |=| ?)

can be formulated to respond to the question, "What was the score of the Red Sox/Yankee game on July 7"? The more restricted syntax makes it easier for the module that executes (responds to) the query to interpret it.

This is a format suitable to applications in which command languages are used to interact with a specific data base, for example, a set of airline data. The system could be designed both to query the data base and to execute commands (reserve, cancel), with respect to the contents of the data base, which represent the real-world objects (passengers, planes, days).

A system similar in design and function to the query/command processor just described, through one more complex in design in some respects, is the system SHRDLU [*23*]. SHRDLU is used to interact with a "world of blocks" (computer representations, not real-world blocks), allowing one to cause events to occur (give commands or requests, i.e., polite commands), and to query the processor about the current status of that world. The essence of the program to manipulate the blocks is a set of decision rules and "plans" indicating what to do in certain situations:

> If the command is to move block x, and there is a block, y, on x, then first remove block y.

The essence of the command world observer is to record the initial "state of the world" and the changes made to it, responding to queries about the world from this data base. Typical examples of commands that might be issued to the blocks world processor, which manipulated the blocks, are

> Pick up the red block.
> Place block A on block B.
> Remove the green block from the red block.

Typical queries to the command world "observer" might be

> Is there anything on block A?
> Is the green block on the red block?

The program is procedural that is, key words trigger the execution of certain procedures. The procedures carry out the tasks and record the results of the activity. This methodology can be applied to the design of command language interpreters, which are also procedural (modular) in nature. The addition of the query capability would make the application capable of providing information about the application area (What files exist? Where are they?), in addition to the ability to execute tasks in that world. These concepts can be applied to the design of command languages that interface with data bases representing and manipulating objects within a computer system, such as files or documents, or real-world objects, such as passengers and flights, and their relationships to one another (reservations, a relationship of degree two, between flights and passengers (customers)).

Applications in natural language processing (NLP) are broader in scope than the design of command languages. Programs have been written to interpret text, referred to as NL comprehension, for example, reading and summarizing newspaper articles [*24*], and generate text, for example, interpreting stock data and writing NL stock reports [*25*]. Command languages are more limited in scope and are designed to carry out some restricted range of tasks, such as editing a file or interacting with an operating system, or, in specific applications areas, recording and responding to queries about bank transactions or reserving seats on a vehicle. However, the techniques of NLP, for example, parsing the text, describing the language (with a grammar), designing the syntax and lexicon for the language, as well as the semantics and rules of action, are issues common to all applications involving some form of NLP [*15, 26–29*]. These issues are also relevant to the design and implementation of programming languages, although the syntax expected by a compiler or interpreter is also relatively more restricted that that of NLP in general [*30, 31*].

Voice Input and Speech Output

Current research and practice is focusing, in some applications, on the use of voice input and speech output [*32*]. Systems have been developed to accept a limited number of commands in a limited domain of discourse (voice input or speech recognition), as well as to respond to a limited set of queries (speech output or speech generation). The development of voice input systems is constrained by the variation in frequencies and the resulting form of words when spoken by different people, or by the same person in different contexts. The phonetic appearance of words and parts of words (phonemes), as well as the transitions between words, varies in the speech of both single individuals, depending on the surrounding text, and across individuals, even in the same context [*33-35*]. However, systems have been developed to accept a limited range of commands from a wide variety of speakers or a more extensive set of words from a limited range of speakers [*36–40*].

Speech output systems have been developed for commercial applications, such as the production of vocal responses to queries made to a data base over the phone, using the phone keys as the means of entering the commands or queries, the development of speech packages to interpret text for the blind, and in research settings [*41, 42*]. One of the questions in the design of speech output systems is the means of generating the output. Systems with predefined outputs can use recorded messages, whereas more flexible systems use speech synthesizers [*43*]. Synthesizing may be done dynamically, by combining appropriate frequencies in sequence, as needed, or statically, by having predefined combinations of frequencies for a limited range of responses. A major issue in the development of speech synthesizers is the quality of the output. Very often it is not of "human quality." Again, however, results can be acceptable in applications suited to voice output, such as those with a limited set of predefined responses, a restricted vocabulary, or a restricted range of users. Systems can also be developed (trained) to the characteristics of a particular user or class of users. In other applications, such as the development of speech output for inter-

pretation of text to the blind, the lesser quality of the output is acceptable because of the criticality of the service provided.

The development of voice input/speech output systems is relevant to the design of both command languages and to command and control systems in general [*44–47*]. Voice input to systems such as editors or operating systems is feasible in systems designed for a limited range of users, because of the limited range and predefinition of the commands. The system could be designed to scan input "text" for key words, using templates of the expected speech patterns (formants) for the commands, and trigger the appropriate processing routines [*48*]. Unrecognized commands would be responded to with an error message or a predefined query. Queries (to the user about his or her intentions) could also be formed dynamically.

The issue of speech output is relevant to the development of query answering systems but not necessarily critical to the development of task-oriented languages (editors, operating systems); it could be used, however, to provide feedback, indicating success or failure of the command, to provide guidance to the user, or in response to a command to print, vocally, for example, the mail messages (with voice input to reply).

Prefix, Postfix, and Infix

An issue that also receives attention in the literature is the use of prefix, postfix, or infix notation [*49, 50*]. The terms prefix, postfix, and infix reflects their usage to describe the position of the operation symbol in arithmetic or logical expressions:

Infix: A |+| B
Prefix: |+| A, B
Postfix: A, B |+

The three expressions receive the same interpretation:

Add A to B

or, more generally, perform the indicated operation on the indicated operands, yielding the appropriate result. In more abstract form, the three formats are described as

Infix: a |o| b
Prefix: o |a, |b
Postfix: a, |b |o

where the operation symbol "o" is to be replaced by any appropriate operation (+, *, and, or, "type"), and the operands of the expression (a, b) are to be replaced by data values of the expected data types (numerical values, the logical values 0 and 1, or the text indicating the name of the file to be typed). In terms of command language design, the operation symbol, or operator, represents the various commands, such as "print" or "substitute" for editors, "dir" (list directory contents), or "type" (type file contents) for operating systems. The operands represent the arguments to the commands (the filename; the target and replacement patterns; *.BAS to select all BASIC programs; -h for "no header").

The commands illustrated earlier, in describing ed, were simplified in form; the commands usually had an implicit argument: "the current line." Only the command to substitute indicated usage of an explicit argument list, such as "pattern" and "new value," indicating the text to be changed (target pattern) and the text with which to replace it (replacement pattern). However, the more general format of command lines is that each command takes one or more arguments, implicit or explicit. The command to print lines 1 through 3 in ed is of the format

1,3p

The placement of the operator (p) after the operands (line numbers 1,3) indicates that this is postfix notation. The function (operation/command) follows the arguments to the function (operands). The same command, using prefix notation, would be

p1,3

with the command placed first. In the examples 1,3p and p1,3, the interpretation of the operand as a single operand ("1,3") or two operands separated by a comma ("1," "3"), is immaterial. However, if one were to implement the command with infix notation, the two-operand interpretation would be used:

1p3

In current practice, prefix or postfix notation is frequently used in designing command languages, and infix is used in creating arithmetic or Boolean expressions in programming languages or query systems. Some research has been done into the effects of prefix or postfix notation on user performance, as well as into other aspects of the interaction with editors [*51, 52*]. Some questions being addressed are whether or not a language designed in one way or another can decrease learning time, increase efficiency of interaction (e.g., increase speed of the overall session), reduce errors (in keying and in the correct execution of the overall task), as well as influence user preference. One hypothesis is that a language design that parallels the structure of imperative statements in one's NL

Pass the salt
Please, pass the salt (prefix)

will have a positive effect on user performance and preference for a particular command language. Another hypothesis is that prior usage with one type of editor (update, prefix) may help or hinder one's learning rate or performance using a subsequent language to perform the same task (ed, postfix), depending on the consistency (match or mismatch) between the two. Another question that arises is whether or not the limited range of commands utilized in many command languages makes any impediment to learning or performance a short-term effect, or whether it can persist over a longer period of time [*53*]. The issues are not only addressed with regard to the use of command languages implemented in computer software but to the use of other devices, such as calculators [*54, 55*].

Lexicon

The issue of prefix versus postfix is a syntactic issue. It concerns the order of the words in a given command, that is, the number and relative placement of the operands. For example, a command such as substitute can have the arguments

1,3s/that/this/g
- 1,3: the range of lines over which the command should operate
- that: the text to be removed
- this: the text to be inserted
- g: a command to change every occurrence of "that" to "this," if there is more than one occurrence. ("Global," over the range of each line; without the "g," only the first occurrence of "that" is changed.)

A distinct issue that arises is the selection of the words to be used for the commands (operators) and arguments (operands). The issue primarily concerns the question of the operators, because the operands are usually the names of objects, for example, filenames, variables, or numeric values, generated by the user of the system. One example of the lexical issue is the usage of the word "cat" to display the contents of a file or a CRT screen when using the UNIX operating system. The usage originated when a routine originally designed to concatenate disk files was found useful to print their contents as well [6,56]. The usage is unnatural and certainly has a surprise, and sometimes annoying effect, when first encountered by users new to UNIX. There is some sentiment for the design of command languages with more "natural" words to name the various operations [57]. Some research has been performed, however, that indicates that users can learn to use even an unnatural dictionary of words (lexicon) in at least a limited context [53]. The same questions of type of effect (performance, preference) and permanency of effect (short term, long term) arise in the lexical arena as in the consideration of syntax. The issue of lexical design, however, concerns the area of semantics (meaningful interpretations), whereas the prefix/postfix issue and other questions concerning the number and form of arguments are concerned with syntax. The design of an artificial language requires both components, as in the case of NL comprehension and generation [34,58], although the task of designing command languages is less comprehensive in scope and the role of semantics less pronounced, because the action taken in response to given commands is usually well defined. defined.

Function Keys and the Mouse

Alternate means of entering commands (to keying of the entire command) are the use of function keys or special input devices such as the mouse or light pen [59,60]. The use of function keys to manipulate a BASIC program (e.g., F1 to list or F2 to run) is a practice familiar to most users of microcomputers. The primary purpose of the function keys is to reduce the time necessary to enter common commands and to reduce the error rate in keying the commands. By substituting a single keystroke for several, the time/error rate is transformed from a variable, dependent on the length of the command, to a relatively constant value, depending on the time it takes to locate and press the appropriate key. The function keys are interpreted

differently by different software packages. Hence, the key that indicates to BASIC that a listing is desired may be interpreted by a screen editor as a command to substitute text (to be subsequently entered) for other text, and by an operating system as a command to list the contents of the current directory.

The use of a mouse or a light pen as an alternative mode to the use of function keys or extended keying is also popular with the users of microcomputers and workstations.

These devices are usually used in conjunction with a menu-selection system (discussed below). The mouse is utilized to position the cursor at a particular item in the menu and clicked to indicate that the command is selected. The operands, if any, are then selected from another menu, representing objects (e.g., files or documents, type fonts, paint palettes) or they are keyed in. The representation of the objects, when they are displayed visually (rather than by name) is often in icon form [61,62]. The treatment of special devices to enter commands and arguments is one aspect of the overall design of workstations and the systems software to interact with them. Other aspects are the type of display [63], amount and arrangement of the display area [64], as well as the work space area [65], the power and flexibility of the functions available [29,66], and the overall design of the user interface [67,68]. The design of the command language and means of issuing the commands is one aspect of the user interface. The design of workstations and the systems software to accompany them is addressed both in the area of human factors [65,69] and in the literature on human/computer (machine) interaction [70,71].

Screen Editors

Although the use of line editors is easier to explain in written form, the use of screen editors is more common in practice. The discussion of screen editors indicates how general the notion of a command language is and exemplifies the joint use of function keys and keyed text. The commands given to a screen editor parallel those given to a line editor (commands to enter, delete, change, and move text), as well as commands to position the cursor on the screen. The commands often use single-function keys (insert and delete on microcomputer keyboards), "arrow" keys on terminals and microcomputer keyboards), or sequences of keystrokes that are interpreted as function keys (e.g., control-d to "move down," i.e., to a subsequent portion of the text in the document). The usability of a screen editor, both in terms of performance and personal preference, is directly related to design of the function keys and/or keystroke sequences. It is also directly related to the type of data display (e.g., single screen area or windows) and the range (power and flexibility) of commands available.

Menu Selection

Although the use of menu selection is not always considered to be an instance of a command language (in fact, is sometimes constrasted with command languages), it is one form of entering commands to a software package. Common examples are the menu systems used in banking applications, such as the automated teller machines (ATM)

1. Withdrawal
2. Deposit
3. Balance inquiry

or those used in data entry applications

1. Add (record)
2. Retrieve (record(s))
3. Delete (record(s))
4. Update (change records)

Menu selection is utilized with casual, that is, infrequent or untrained users (the ATM application), as well as in some applications with dedicated or frequent and trained users (the data entry application). The use of the menu system substitutes entry of a number or letter for the actual command. This obviates the need for the table lookup or other means of parsing discussed above. The code (responses) can be used directly to select the appropriate processing routines. The entry of operands is handled by subsequent menus

1. From credit card?
2. Checking?
3. Savings?

or other means of interaction, such as prompts for given data field values

Social security number →

or selection from a set of icons. Recent research has centered on variations in the mode of presenting menus, for example, options embedded in the text [72]. The use of menu-based systems eliminates the need for the user to remember the options (commands) available. This is its purpose in applications involving even dedicated users.

The one drawback to menu selection is the length of time necessary to complete a transaction. The use of an ATM may take up to three or four display/response subtransactions (dialogue pairs) [73] in order to complete a single request (overall transaction). This lengthens the time of human/computer interaction, consequently lowering the service rate and, if telecommunications are involved, increasing the load on the communication channel (lines). For these reasons, in frequently used systems, in which the service rate is a critical design feature and communications costs are to be minimized, command mode is used.

Command Mode

Command mode is the creation and entry of the commands (operation and operands) by the actual user of the system. The actual user may or may not be the end user or "client" (the beneficiary or recipient of the transaction's results). In systems such as editors or operating systems, the end user or client and the manipulator of the system are usually identical. In systems operated as part of an enterprise, such as reservations or banking, or data base searching (especially with bibliographic data bases), the

actual manipulator, teller, reservation clerk, or reference person is often distinct from the client and is referred to as an intermediary or third-party user [74].

A good example of the use of command mode is given by Martin [8], in one of the earlier texts in the design of human/computer dialogue (languages). The example concerns the design of an airline reservation system. A typical command might be the following:

A31JanPghOrd700A

Each portion of the command has a specified meaning. For example, in the code illustrated above, the meaning of the respective symbols is as follows:

A—What is available?
 on Jan 31
 from Pittsburgh to O'Hare (Pgh,Ord)
 departing in the vicinity of 7 a.m.

The system responds to the command either with the requested information (e.g., in response to a query for a list of flights and seat availability) or with the requested transaction (e.g., updating the data base to reflect the reservation of a set of seats). A particular transaction (the equivalent of an entire editing session or an entire interaction with the operating system) may take several request/response pairs, but the total time of interaction is shortened considerably, as compared to menu-based systems used to accomplish the same task. It is because of the efficiency of interaction that experienced users of a system often prefer command mode to a menu-based system. However, menu-based systems are usually easier to learn. Hence, many systems are designed with a menu system for beginning users and a command mode for experienced users, with some means of selecting between the two.

Operating System Languages

The command languages used to interact with an operating system share several common functions:

- create a file
- list the contents of a directory
- list the contents of a file
- rename a file
- copy a file
- delete a file
- create a new directory
- delete a directory
- move a file from one directory to another
- select among other software utilities, such as a sort, compilers, assemblers, application routines, and packages

Hence, the form of the basic set of commands is relatively uniform across many systems. For example, some of the commands relative to file handling on the UNIX system are the following:

ls—list the names of files and subdirectories (child directories) for a given directory
cat—list the contents of a text file on the screen
lpr—print the contents of a file on the specified or default printer
mkdir—create a subdirectory to the current directory
chdir—change my "location" from the current directory to the specified directory
rm—remove (delete a file

The corresponding commands to MS-DOS are

-dir
-type
-print
-mkdir
-chdir
-delete

The commands may take implicit arguments

ls (implicit argument is "this directory")

explicit but optional arguments

ls -l (-l indicates a long listing, not only indicating names but creation, date, protection codes, and so forth)

or explicit and nonoptional arguments (at least for successful execution)

cat filename
type filename

The commands and their legal formats are explained in the manual accompanying each system and in the various textbooks [*6, 75–80*].

Error Messages

As the parenthetical comment above indicates, the execution of a given command is not always successful. When the execution of the command is unsuccessful due to the inability of the system to interpret the command, an error message or other symptom of nonsuccess is usually given [*81–83*]. The phrase "is usually given" is used because it is sometimes impossible to identify an error, due to the design of the system. It is also the case that errors may be identified, but a correct diagnosis of the cause of the error may be difficult to provide.

One example is the usage of cat, the command to print a file in UNIX. If one omits the argument (filename) to cat, the system merely types a prompt, waiting for input from the screen, which it will then echo to the screen. This is not usually the response desired by someone who omits the filename, but it is a legal response to this command in UNIX. The usual UNIX response to an error is a question mark or a question mark accompanied by a short message, but the cat command issued in isolation is

"interpretable," and, hence, does not generate an error message. The choice of an error message consisting solely of a question mark has, in itself, been the subject of controversy. Expert users tend to like the succinctness, whereas novices tend to dislike the omission of specificity. (The differences between novice and expert behavior has been studied, both with respect to command languages [*49*] and programming languages [*84–86*] and with respect to problem solving in general, such as chess playing [*87*]).

The design of appropriate error messages is a nontrivial task, as it is often difficult for the command interpreter to diagnose the exact cause of an error. For example, the omission of the space (or comma) between the two file names "File1" and "File2" in the command:

type File1File2

would generally result in the message

File not found

rather than

comma or space missing

The reason for the misdiagnosis is that the string

File1File2

is a legal filename, though probably one that is nonexistent, given the intent of the command, which is to print the contents of two files, File1 and File2, but has simply been mistyped. The problem is analogous to that faced by the error message routines of compiled programs: Because the object code of the compiled program no longer has a close connection to the source program, "abstract error messages," such as the following, are reported at execution time

subscript out of range,
location hex 4AE5

rather than

subscript out of range on line 10

The problem is less severe for command language interpreters, however, because they are, as the name indicates, interpreters and, hence, have a closer relationship (an immediate relationship) to the origination of the command than the monitor of object code execution does.

Cognitive Models

More recent research in the design of command languages and the systems software that accompanies them has centered on the overall cognitive model that a user has of the task to be performed [*88*]. A person interacting with an editor wishes to create a document or a computer program. The

creation of the document (or program) is the primary or final goal of the interaction. The attainment of the primary goal involves the setting of subgoals: Create the introductory paragraph, change sentence three, locate the closing statement. The task goals are stated at the level of the task/cognitive domain at varying levels of "granularity."

The cognitive domain can be throught of as a problem space, wherein the basic task is to transform the current state of an object or set of objects in the real world into a goal state, the finished product. The operations to accomplish the transformations are the operators in the cognitive domain [89]. The system also represents a problem space but the objects and operations of the system do not, in the usual case, match the operations and operands in the real-world space exactly. Hence, a translation must be made between the real-world task (create introductory paragraph, rewrite line 3) to those of the system space (insert text, move to line 3, substitute). It is the task of the user to translate the overall goals and subgoals of the real-world task into the specific tasks and commands of the system, for example, "ed document" or "p1,3." The ease with which the translation or mapping is made depends on the closeness of the system design and implementation to the cognitive or task domain. The level at which the mapping is made can vary, and the effect is seen in the flexibility or lack of it in the system. Systems designed at a very "high" level allow interaction with "objects," such as documents, paragraphs, reservations, and blocks of computer code [90], that is, the elements of the real-world task domain, whereas systems designed at a more basic level cause one to interact with "lines," "positions," and "files," that is, with objects in the system domain. The higher level commands, such as "edit document," allow one to translate the real-world tasks into the system tasks more readily (with less need for training) but allow for less flexibility of application. An editor designed to deal with documents cannot readily be used for creating programs, and vice versa, whereas a general-purpose editor can be utilized for both. The lower-level commands require more training but generally give flexibility. Alternatively, one could build a special-purpose interface to a general-purpose system, allowing the user to select between the two.

The design of systems at a conceptual level, as well as the training of users to map between the cognitive/world domain and the systems domain, is studied both in the area of cognitive science and in the area of human/machine interaction. The interface area is the area where the mapping takes place [91–93].

SUMMARY

A command language interpeter is a device (software or hardware) for accepting, interpreting, and executing commands relevant to a particular application. A command language is the language used to interact with the interpreter. Typical applications are interactions with an operating system, editor, or transaction-oriented system. Issues of concern in the design of command languages are the type of language to use, highly artificial or "natural," the design of the syntax and lexicon of the language, and the mode of entering commands. Research is being carried out in the areas of language design and mode of I/O, as well as into the overall design of the system. Some measures of performance are ease of interaction, success in completion of the task, speed of interaction, error rate, and personal preference.

Although command languages are usually discussed with respect to command mode, both command mode and menu selection are means of entering commands to a command language interpreter. Menu selection tends to be used with untrained or infrequent users, although this is not always the case; command mode is utilized with trained users, because one cannot learn the command formats (placement of operators, operands, and options) without some learning experience. Because the desire to interact more efficiently often follows the acquisition of skill with a particular system, many applications are equipped with both modes, with some means of choosing between the two.

The research into the design of effective languages and devices for human/computer and human/machine interfaces is receiving increasing attention in the areas of cognitive science, human factors, and artificial intelligence, as well as in the design of commercial applications. One area of interest is in the use of NL (or flexible language and format) for entering the commands. As indicated above, the parsing of a command to a command interpreter is relatively simple compared to the parsing of NL text. The problem consists primarily in the isolation of the command and the determination of the number and type of the arguments given. However, there are indications that the choice of appropriate syntax and terminology can facilitate learning, performance, and preference and that inadequate designs can hinder the same.

Other areas of concern are the overall design of the interface area (the display/control area), the mode of entering commands, and the responses to these, as well as the physical layout of the work space. A recent area of concern has been the cognitive domain of the user with respect to the task (problem) space. The latter is the fundamental concern in the design of command language interpreters, because the interpreter is simply a means of accomplishing some set of tasks with a particular set of software and hardware. The task of the user, designer, and person implementing the system is to translate the tasks in the problem space into the operations, commands, and arguments (operands) of the system space. The user interface is the area at which the mapping real-world/system takes place. Implementation issues are addressed in the literature concerning the particular applications, such as editors, operating systems, and applications programs, both with respect to the code written and the overall design of the user interface.

REFERENCES

1. Roger R. Flynn, "Assembly Language", in *Encyclopedia of Microcomputers*, Vol. 1, Marcel Dekker, New York, 1987.
2. ACM (Special issue: Papers from the Conference on Human Factors in Computing Systems.) *Commun. ACM, 26*(4) (April 1983).
3. ACM (Special issue: The Psychology of Human Computer Interaction.) *Comput. Surv., 13*(1) (March 1981).
4. Albert Badre, and Ben Shneiderman, eds., *Directions in Human/Computer Interaction*. Ablex, Norwood, NJ, 1982.
5. *Int. J. Man-Mach. Stud.*, (Special Issue: The Semantics and Syntax of Human-Computer Interaction.) Vol. 15, 1981.
6. Brian W. Kernighan and P.J. Plauger, *Software Tools*, Addison-Wesley Reading, MA, 1976.
7. Edward Yourdon, *Design of On-Line Computer Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1972.
8. James Martin, *Desgin of Man-Computer Dialogues*, Prentice-Hall, Englewood Cliffs, NJ, 1973.
9. William M. Newman and Robert F. Sproull, *Principles of Interactive Computer Graphics*, 2nd ed., McGraw-Hill, New York, 1979.
10. Charles T. Meadow, *Man-Machine Communication*. Wiley, New York, 1970.
11. Thomas P. Moran, "The Command Language Grammar: A Representation for the User Interface of Interactive Computer Systems," *Int. J. Man-Mach. Stud.*, *15*(1), 3–50 (July 1981).
12. Thomas W. Malone, "Heuristics for Designing Enjoyable User Interfaces: Lessons from Computer Games, in *Schneider, eds., Human Factors in Human Computer Systems* (Thomas and Schneider, eds.), Ablex, Norwood, NJ, 1984, pp. 1–12.
13. E. Bertino, "Design Issues in Interactive Interfaces in Computing." *User Interfaces, 3*(1), 37–53 (February 1985).
14. Joseph Weizenbaum, "ELIZA—A Computer Program for the Study of Natural Language Communication between Man and Machine," *Commun. ACM, 9*(1), 36–45 (January 1966).
15. Terry Winograd, *Language as a Cognitive Process: Volume I: Syntax*. Addison-Wesley, Reading, MA, 1983.
16. American National Standards Institute, (ANSI), *American National Standard: COBOL (ANSI X3.23-1968)*. ANSI, New York, 1969.
17. T. C. McMillan and B. P. Norman, "Command Line Structure and Dynamic Processing of Abbreviations in Dialogue Management." *Interfaces in Comput.*, *3*, 249–257 (1985).
18. Charles Rich and Richard C. Waters, *Readings in Artificial Intelligence and Software Engineering*. Morgan Kaufmann Los Altos, CA, 1986.
19. James A. Craig, Susan Berezner, Homer Carney, and Christopher Long, "DEACON: Direct English Access and Control," in *Proceedings of the AFIPS 1966 Fall Joint Computer Conference*, Spartan Books, New York, 1966, pp. 365–380.
20. Bert F. Green, Alice K. Wolf, Carol Chomsky, and Kenneth Laughery, "BASEBALL: An Automatic Question Answerer, in *Computers and Thought* (Edward A. Feigenbaum and Julian Feldman, eds.), McGraw-Hill, New York, 1963, pp. 207–216.

21. Herbert H. Clark and Eve V. Clark, *Psychology and Language: An Introduction to Linguistics*, Harcourt Brace Jovanovich, New York, 1977.
22. Bertram Raphael, *The Thinking Computer: Mind Inside Matter*. W.H. Freeman, San Francisco, CA, 1976.
23. Terry Winograd, *Procedures as a Representation for Data in a Computer Program for Understanding Natural Language*, Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, MA, 1971.
24. Gerald de Jong, "An Overview of the Frump System," in *Strategies for Natural Language Processing*, (Lehnert and Ringle, eds.), Lawrence Erlbaum Associates, Hillsdale, NJ, 1982.
25. Karen Kukich, *Knowledge-Based Report Generation: A Knowledge Approach to Natural Language Report Generation*, Ph.D. Thesis, University of Pittsburgh, Pittsburgh, PA, 1983.
26. Wendy G. Lehnert and Martin H. Ringle, *Strategies for Natural Language Processing*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1982.
27. R. F. Simmons, J. F. Burger, and R. E. Long, An Approach toward Answering English questions from Text, in *Proceedings of the AFIPS 1966 Fall Joint Computer Conference*, Spartan Books, New York, 1966, pp. 357–363.
28. Robert J. K. Jacob, "Using Formal Specifications in the Design of a Human-Computer Interface," *Commun. ACM, 26*(4), 259 (April 1983).
29. Phyllis Reisner, "Formal Grammar as a Tool for Analyzing Ease of Use: Some Fundamental Concepts," in *Human Factors in Human Computer Systems*, (Thomas and Schneider, eds.), Ablex, Norwood, NJ, 1984, pp. 53–78.
30. Alfred V. Aho and Jeffrey D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, MA, 1979.
31. R.E. Berry, *Programming Language Translation*. Ellis Horwood, New York, 1981.
32. James L. Flanagan, "Computers that Talk and Listen: Man-Machine Communication by Voice." (Dixon and Martin, eds.), 1979, pp. 4–14.
33. Colin Cherry, *On Human Communication*, 2nd ed., M.I.T. Press, Cambridge, MA, 1966.
34. Bonnie Nash-Weber, "The Role of Semantics in Automatic Speech Understanding," in *Representation and Understanding*, *Studies in Cognitive Science*, (Daniel G. Bobrow and Allan Collins, eds.), Academic Press, New York, 1975.
35. L. R. Rabiner and R. W. Schafer, *Digital Processing of Speech Signals*, Prentice-Hall, Englewood Cliffs, NJ, 1978.
36. S. R. Hyde, "Automatic Speech Recognition: A Critical Survey and Division of the Literature," in *Automatic Speech and Speaker Recognition*, (Dixon and Martin, eds.), IEEE Press, NY, 1979.
37. Avron Barr and Edward A. Feigenbaum, "Understanding Spoken Language," in *The Handbook of Artificial Intelligence*, Volume I, William Kaufmann, Los Altos, CA, 1981, pp. 323–361.
38. John Gould, John Conti, and Todd Hovanqecz, "Composing Letters with a Simulated Listening Typewriter," *Commun. ACM, 26*(4) (April 1984).
39. A. Giordana and L. Saitta, "Discrimination of Words in a Large Vocabulary Using Phonetic Descriptions," *Int. J. Man-Mach. Stud. 24*(5), 453–473 (May 1986.
40. B. E. Pay and C. R. Evans, "An Approach to the Automatic Recognition of Speech," *Int. J. Man-Mach. Stud. 14*(1), 13–27 (January 1981).

41. R. Kurtzweil, "Kurzweil Talking Terminal Announced; First Unlimited Vocabulary. The Kurzweil Report, Kurzweil Computer Products, Cambridge, MA, 1979.
42. Media Dimensions, Inc., *Proceedings, Speech Technology '85: Voice Input/Output Applications, Show and Conference*, New York, April, 1985.
43. Paul R. Michaelis and Richard H. Wiggins, "A Human Factors Engineer's Introduction to Speech Synthesizers," in *Directions in Human Computer Interaction*, (Albert Badre and Ben Shneiderman, eds.), Ablex, Norwood, NJ, 1982, pp. 149–178.
44. N. Rex Dixon and Thomas B. Martin, *Automatic Speech and Speaker Recognition*, IEEE Press, New York, 1979.
45. Wayne A. Lea, ed., *Trends in Speech Recognition*, Prentice-Hall, Englewood Cliffs, NJ, 1980.
46. Peter C. Bell, Genevieve Hay, and Y. Liang, "A Visual Interactive Decision Support System for Workforce (Nurse) Scheduling." *INFOR*, *24*(2), 134–135 (May 1986).
47. D. W. Connoly, *Voice Data Entry in Air Traffic Control*, FAA-NA-79-20, 1979.
48. Richard W. Christiansen and Craig K. Rushforth, "Detecting and Locating Key Words in Continuous Speech Using Linear Predictive Energy," in *Automatic Speech and Speaker Recognition*, (Dixon and Smith, eds.), IEEE Press, New York, 1979, pp. 211–217.
49. J. M. Cherry, *Command Languages: Effects of Word Order on User Performance*, Ph.D. Thesis, University of Pittsburgh, 1983.
50. J. M. Cherry, "An Experimental Evaluation of Prefix and Postfix Notation in Command Language Syntax." *Int. J. Man-Mach. Stud.*, *24*(4), 364–374 (April 1986).
51. T. L. Roberts and T. P. Moran, "The Evaluation of Text Editors: Methodology and Empirical Results," *Commun. ACM*, *26*, 265–283 (1983).
52. S. K. Card, W. K. English, and B. J. Burn, "Evaluation of Mouse, Rate-Controlled Isometric Joystick, Step Keys and Text Keys for Text Selection on a CRT, *Ergonomics*, *21*, 601–613 (1978).
53. Susan T. Dumais and Thomas K. Landauer, "Psychological Investigations of Natural Terminology for Command and Query Languages," in *Directions in Human Computer Interaction* (Albert Badre and Ben Shneiderman, eds.), Ablex, Norwood, NJ, 1982, pp. 95–109.
54. F. G. Halasz, "Mental Models, and Problem Solving in Using a Calculator," Ph.D. Thesis, Stanford University, Stanford, CA, 1984.
55. Richard M. Young, "The Machine Inside the Machine: Users' Models of Pocket Calcuations," *Int. J. Man-Mach. Stud.*, *15*, 51–85 (1981).
56. Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1978.
57. Robert Wilensky, Yigal Arens, and David Chin, "Talking to UNIX in English: An Overview of UC," *Commun. ACM*, *27*(6), 574–593 (June 1984).
58. L. Stephen Coles, "Syntax Directed Interpretation of Natural Language," in *Representation and Meaning, Experiments with Information Processing Systems*". (Herbert A. Simon and Laurent Siklossy, eds.), Prentice-Hall, Englewood Cliffs, NJ, 1972, pp. 211–287.
59. S. K. Card, W. K. English, and B. J. Burn, "Evaluation of Mouse, Rate-Controlled Isometric Joystick, Step Keys and Text Keys for Text Selection on a CRT, *Ergonomics*, *21*, 601–613 (1978).

60. John Karat, James E. McDonald, and Matt Anderson, "A Comparison of Menu Selection Techniques: Touch Panel, Mouse and Keyboard," *Int. J. Man-Mach. Stud.*, *25*(1), 73–88 (July 1986).
61. D. Gittens, "Icon-based Human Computer Interaction," *Int. J. Man-Mach. Stud.*, *24*(6), 519–543 (June 1986).
62. J. Gutknecht, "Concepts of the Text Editor LARA," *Commun. ACM*, *28*(9) (September 1985). *Comput. Practices*, 942–960.
63. B. G. Pearce, ed., *Health Hazards of VDTs*? Wiley, Chichester, Great Britain, 1984.
64. Kent L. Norman, Linda J. Weldon, and Ben Schneiderman, "Cognitive Layouts of Windows and Multiple Screens for User Interfaces," *Int. J. Man-Mach. Stud.* *25*(2), 229–248 (August 1986).
65. Ernest J. McCormick and Mark S. Sanders, *Human Factors in Engineering and Design*, 5th ed., McGraw-Hill, New York, 1982.
66. M. V. Mason, "Adaptive Command Prompting in an On-Line Documentation System," *Int. J. Man-Mach. Stud.*, *25*(1), 33–51 (July 1986).
67. Mike L. Schneider, "Ergonomic Considerations in the Design of Command Languages," in *Human Factors and Interactive Computer Systems*, (Y. Vassiliou, ed.), Ablex, Norwood, NJ, 1984, pp. 141–161.
68. J. N. J. Richards, H. E. Bez, D. T. Gittins, and D. J. Cooke, "On Methods for Interface Specification Design," *Int. J. Man-Mach. Stud.*, *24*(6), 545–568 (June 1986).
69. Barry H. Kantowitz and Robert D. Sorkin, *Human Factors: Understanding People-System Relationships*. Wiley, New York, 1983.
70. John C. Thomas and Michael L. Schneider, eds., *Human Factors in Computer Systems*, Ablex, Norwood, NJ, 1984.
71. Yannis Vassiliou, ed., *Human Factors and Interactive Computer Systems*. Ablex, Norwood, NJ, 1984.
72. Larry Koved and Ben Schneiderman, "Embedded Menus: Selecting Items in Context," *Commun. ACM*, *29*(4), 312–317 (April 1986.
73. James Martin, *Systems Analysis for Data Transmission*. Prentice-Hall, Englewood Cliffs, NJ, 1972.
74. F. W. Lancaster and E. G. Fayen, *Information Retrieval On-Line*, Melville, Los Angeles, CA, 1973.
75. Kaare Christian, *The UNIX Operating System*, Wiley, New York, 1983.
76. Douglas Comer, *Operating System Design: The Xinu Approach*, Prentice-Hall, Englewood Cliffs, NJ, 1984.
77. *Bell Syst. Tech. J.*, *57*(6), Part 2 (July–August 1978).
78. Henry M. Levy and Richard H. Eckhouse, *Computer Programming and Architecture: The Vax-11*, Digital Press, Bedford, MA, 1980.
79. Peter Norton, *Programmer's Guide to the IBM PC*. Microsoft Press, Bellevue, WA, 1985.
80. Milan T. Milenkovic, *Operating Systems: Concepts and Design*, McGraw-Hill, New York, 1987.
81. Donald A. Norman, "Design Rules Based on Analyses of Human Error," *Commun. ACM*, *26*(4) (April 1983).
82. P. J. Brown, "Error Messages: The Neglected Area of the Man/Machine Interface?" *Commun. ACM*, *26*(4) (April 1983).
83. Ben Shneiderman, "System Message Design: Guidelines and Experimental Results," in *Directions in Human Computer Interaction*, Ablex, Norwood, NJ, 1982, pp. 55–78.

84. Elliot Soloway and Sitharama Iyengar, eds., *Empirical Studies of Programmers: Papers Presented at the First Workshop on Empirical Studies of Programmers*, Ablex, Norwood, NJ, 1986.
85. Susan Marie Wiedenbeck, *Some Determinants of Skilled Performance in Programming*, Ph.D. Thesis, University of Pittsburgh, Pittsburgh, PA, 1984.
86. Susan Wiedenbeck, "Novice/Expert Difference in Programming Skills," *Int. J. Man-Mach. Stud.*, *23*, 383–390 (1985).
87. Allen Newell and Herbert A. Simon, *Human Problem Solving*, Prentice-Hall, Englewood Cliffs, NJ, 1972.
88. Mary S. Riley, "User Understanding," in *User Centered System Design: New Perspectives on Human Computer Interaction*, (Norman and Draper, eds.), Lawrence Erlbaum Associates, Hillsdale, NJ, 1986, pp. 157–185.
89. S. Card, T. Moran, and A. Newell, "The Goms Model of Manuscript Editing," in *The Psychology of Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1983, pp. 139–191.
90. Richard C. Waters, "The Programmers's Apprentice: A Session with KBEmacs," in *Readings in Artificial Intelligence and Software Engineering*, (Rich and Waters, eds.), Morgan Kaufman, Los Altos, CA, 1986.
91. S. Card, T. Moran, and A. Newell, *The Psychology of Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1983.
92. P. J. Barnard, N. V. Hammond, J. Morton, and J. B. Long, "Consistency and Compatibility in Human-Computer Dialogue," *Int. J. Man-Mach. Stud.*, *15*(1), 87–134 (July 1981).
93. A. A. Clarke, "A Three-Level Human Computer Interface Model," *Int. J. Man-Mach. Stud.*, *24*(6), 503–517 (June 1986).

ROGER R. FLYNN

COMMODORE INTERNATIONAL LIMITED

Commodore International Limited is a fully integrated manufacturer of advanced microcomputer systems, semiconductor components, consumer electronics products, and office equipment. Manufacturing facilities are located in North America, Europe, and the Far East. Marketing is worldwide. Research is devoted primarily to the development of new products using solid-state integrated circuitry, computer technology, and consumer electronics.

COMPUTERS

Computers are at the heart of Commodore's range of products. As the market has grown, so has Commodore's product line grown to meet the differing needs of its users.

Home and Personal Computers

In the home computer sector, the Commodore 64 has received acclaim in leading computer magazines as "The Home Computer of the Year." During the 1984 fiscal year the Commodore 64 also became the top-selling microcomputer in the world. The Commodore 16 and Plus/4 were introduced in time for the 1984 Fall and Christmas seasons.

Educational Computers

In Germany, where Commodore sells an estimated 40 percent of school microcomputers, a special version called the Commodore 4064 has been introduced and is selling well. Commodore intends to continue developing this market, and believes several of its newly introduced products will be very successful in the educational market.

Business Computers

Commodore's business computer sales have been primarily concentrated outside the United States. The mainstays of the product range have been the traditional CBM 8032 and 8096 models. The latter range has been upgraded with the introduction of the CBM 8296 which includes integral dual-disk drives. Because of the large installed base of its predecessors and the extensive local language software available, the CBM 8296 has been well received in Europe.

In 1985 Commodore introduced the Amiga personal computer. The standard configuration of the Amiga includes 256K of random-access memory (RAM), internally expandable to 512K, with 192K of the writable control store, an 89-key keyboard with numeric keypad, cursor, and special function keys, 80 by 25 line text display, maximum 640 by 400 resolution, palette of 4,096 colors, parallel, serial, and second drive ports, three video

and two audio ports, a full 68000 bus expansion, two-button mouse controller, built-in 3.5 inch 880K floppy disk drive, two joystick ports, plus text-to-voice and music synthesis capability. Bundled software includes Amiga DOS, ABasiC, and Amiga Tutor. The Amiga has a suggested retail price of $1,295. By using an available option, the Amiga can run IBM PC-compatible software packages such as Lotus 1-2-3, dBase III, and WordStar in either 3.5 or 5.25 inch format.

Commodore is also developing a UNIX-compatible machine. The machine is multiuser and multitasking with 16-bit architecture at an advanced level and will use a Commodore-developed interface Computer Universal Shell (CUSH). These new business machines were introduced into distribution in 1985 and have considerably strengthened Commodore's overall offerings in this sector of the microcomputer market.

PERIPHERALS

Now that Commodore has the largest installed base of any microcomputer manufacturer worldwide, there is an ever-increasing potential market in peripheral sales. This is especially true in the home market sector where the components of a complete computer system are more often bought one piece at a time starting with the main computer. For example, it is estimated that only 10 percent of consumers buy a printer with their initial home computer purchase.

Accordingly, Commodore has produced a range of "intelligent" peripherals that do not generally use any of the main computer's memory to operate and can be simply connected without costly special interfaces. This is cost-effective advantage which assures Commodore a high percentage of the peripheral sales that complement out computers. During the 1984 fiscal year, peripheral sales accounted for 35.9 percent of business. This compares to 18.4 percent in fiscal year 1983. The same approach has enabled Commodore to offer users a very cost-effective overall computer system which increases its attractiveness to new purchasers.

A Full Range of Disk Drives and Printers

Commodore has complemented its broad variety of computers with two prime ranges of peripherals marketed for home and personal computers—those compatible with the VIC 20, Commodore 16, Commodore 64, and Plus/4. For business computers, Commodore markets a line of high-quality IEEF peripherals. In addition, a full range of printers is offered for all of its computers.

A Range of Other Peripherals

Commodore offers a wide range of other peripheral products, including cassette drive units, joysticks, and paddles. A range of low-price modems is also offered for the growing telecommunications market where Commodore's strong market position has made telecomputing accessible to the masses. The introduction of the Auto-modem has added auto answer/auto dial capabilities to its telecommunications line.

A major value item which has proven extremely popular is the color monitor. With the growth in popularity of computers, the consumer is beginning

to prefer a video monitor to a television set. Commodore has monitors with built-in special circuitry to enhance the clarity of computer video input. This makes Commodore monitors particularly attractive to the installed base of users.

In addition to our general range of products, some of Commodore's regional operating companies offer specialized peripherals to meet their particular market needs. These include, for example, an input/output controller and plotter in the German market where Commodore has a strong presence in scientific use. Commodore is also evaluating, for future inclusion in our product line, peripherals such as touch screens and mouse controllers.

SOFTWARE

In 1983, Commodore made a major commitment to the development of high-quality software with the establishment of the Commodore Software Division. During the 1984 fiscal year Commodore software sales were $82 million of our total sales compared to $63 million in fiscal 1983.

One of the most important achievements of 1984 was the development of a comprehensive range of leading software products that have significantly enhanced the attractiveness of our hardware. These software products cover a full spectrum from productivity, business, education, communication, programming, and entertainment to art and music.

Commodore has a strong range of software products and continues to concentrate on providing innovative productivity software products for the home and office and new products which incorporate speech. In addition to the entertainment value of recreational software, speech synthesis augments the capabilities of the educational products.

Operating Systems

In 1984, Commodore licensed a key operating system to be used in future business products; the UNIX-compatible COHERENT system. Commodore believes this system, in addition to the licensed MS/DOS system, will enable them to create a much stronger presence in the business sectors of the market.

MANUFACTURING

Commodore continues to strive to maintain its position as a high-volume cost-effective producer. Its products are designed with volume production in mind. Buying is done on an international basis in order to achieve the most effective pricing. This requires a strong logistics control operation which has been reinforced at our Far East facilities—the hub of Commodore's manufacturing activities.

Manufacturing is done in three stages: (1) semiconductor wafer fabrication, (2) the semiconductor packaging and board assembly operations, and (3) the final assembly operations.

Semiconductor Wafer Fabrication

This initial phase of manufacturing operation is done primarily in North America where Commodore has invested heavily over recent years. This

investment ensures our own supply of key components. In 1984 our five-inch wafer fabrication facility at Costa Mesa, California became fully operational and is one of the most advanced in the industry. The other major semiconductor facility is in Norristown, Pennsylvania.

Prime Assembly Operations

While demand for many products sometimes exceeded supply in 1984, Commodore's principal production operations have substantially increased volume. The key to this volume increase has been expanded Hong Kong operations which now control Far East manufacturing operations under a centralized coordinated management. In addition to the 220,000 square foot facility at Kwai Chung Center in Hong Kong, which became operational in 1984, Commodore has established a major operation in Taiwan which will add considerably to manufacturing capacity. In addition, a joint venture with Mitsumi Electric Company in Japan is now producing disk drives in volumes that exceeded initial expectations.

Final Assembly Operations

Commodore's assembly operations are located near its major markets in order to minimize freight costs and to enable any differences in local technical requirements to be met by those most familiar with them. This also gives Commodore the greatest flexibility in adjusting the final production processes to meet any changing market needs. These locations are in West Chester, Pennsylvania; Braunschweig, West Germany; and an additional final assembly operation in Corby, United Kingdom, which started production in August 1984.

MARKETING

Commodore computers are now sold in more retail outlets worldwide than any other computer, and its international distribution has continued to grow and strengthen in the last year. In Europe, where we have traditionally been strong in both office and home sectors, several steps were taken to strengthen our position. These included the integration of joint ventures in Denmark, Norway, and Holland into fully owned subsidiaries of Commodore International.

Geographic Diversification

One of the keys to Commodore's success is that it has proficiently combined an international marketing orientation with local geographic specialization. In over 60 countries worldwide Commodore employs talented managers, knowledgeable in local markets, to handle operations from distribution and advertising to software development in local languages. This geographic diversification enables computers to be sold through several thousand outlets throughout the world.

TECHNOLOGY

Commodore's investment in technology development has supported its product growth and cost competitiveness. Commodore believes the depth of its technical strengths is one of its greatest assets.

In recent months Commodore has been successful in substantially increasing its engineering resources. They have opened an additional research and development facility in the heart of Silicon Valley in California. Particular emphasis will be made in computer-aided design, where Commodore is currently embarked upon a multimillion dollar investment program. This will considerably accelerate future development. In particular, it will enable Commodore to run numerous simulations while designing circuits to maximize production yields incorporating its own process rules prior to introduction into large-scale production.

Semiconductor Development

Commodore has continued to invest strongly in semiconductor operations. While the company has developed a number of special proprietary products to enhance the features of existing products, it has also been involved in licensing major devices from other companies. Commodore's development in these instances is focused upon enhancing these products in order to maximize tha overall cost effectiveness of the computer systems. this is accomplished through the development of our own proprietary chips to minimize the overall component count and cost per computer.

WORLDWIDE DISTRIBUTION

One of Commodore's principal strengths is its worldwide retail distribution. Creating mass distribution for home computers is one of Commodore's most salient achievements. The number of outlets carrying Commodore home computers has expanded from fewer than 3,000 stores in 1981 to nearly 35,000 worldwide at the end of fiscal 1984.

Commodore computer systems are now sold in the world's most successful mass retail outlets, specialty computer stores, small retail chains, and catalogue showcases. With this combined distribution, Commodore makes affordable, high-performance computers easily accessible to consumers throughout the world. In order to facilitate this expanded distribution, Commodore has continued expansion in manufacturing and assembly. Most notably, the new installations at Corby in the United Kingdom, Kwai Chung Center in Hong Kong which became operational during fiscal year 1984, will allow Commodore to better serve its growing distribution. In addition, Commodore has recently established a major operation in Taiwan which will further expand the manufacturing capabilities of Commodore International.

BACKGROUND

1958 Founder, Jack Tramiel, establishes portable typewriter company in Toronto, Canada.

1960 Commodore Business Machines, Inc. established in United States.

1962 Commodore acquires office machine manufacturing facility in Berlin, West Germany. Products include desk and portable typewriters as well as electromechanical adding machines.
1965 Large manufacturer of office furniture in Scarborough, Ontario acquired. Irving Gould, Toronto attorney and financier, joins company and provides capital for growth.
1966 Commodore Electronics, Ltd. incorporated in Canada.
1967 Commodore expands business machine line with electronic calculators made in Japan. Company ceases production of electromechanical adding machines.
1968 Commodore opens California office in "Silicon Valley."
1969 Using Texas Instruments semiconductors, work begins on electronic calculators.
1970 Commodore introduces the C108—the first hand-held calculator made in the United States.
1974 Commodore expands its line of calculators from simple four-function machines to memory machines, scientific machines, and keyboard-programmable models. The company lists on the American Stock Exchange and triples initial stock price in first year.
1975 Commodore begins R&D and manufacture of liquid crystal displays. Begins product assembly operations in Hong Kong.
1976 MOS Technology, designer and manufacturer of semiconductors, is acquired. The Norristown, Pennsylvania-based developer of the 6502 microprocessor contributes to Commodore's plans to vertically integrate its engineering and production capabilities. The company also becomes officially known as Commodore International Ltd. KIM-1 introduced.
1977 Commodore introduces PET (Personal Electronic Transactor), a low-cost, stand-alone computer, a forerunner of today's personal computer market. The machine's advanced 6502 microprocessor technology becomes the basis for other manufacturer's personal computers.
1978 PET 4000 Series introduced. Consumers gain more powerful processing capabilities with series 16K and 32K read-only memory (ROM) models. Commodore continues its vertical integration plans with the acquisition of Frontier, a Los Angeles-based maker of semiconductors that complement the 6502 microprocessor.
1979 The company acquires Micro Display System, a Dallas-based manufacturer of liquid crystal display systems.
1980 Commodore unveils the CBM (Commodore Business Machines) 8032 and 8096 business computers with a wide range of business and professional applications programs. VIC (Video Interface Computer) 20 previewed at National Computer Convention and officially introduced at Seibu Department Store, Tokyo. Because of its ability to work on a color or black and white television screen as well as a monitor, the VIC 20 becomes the mainstay of the world's home computer market.
1981 Commodore opens manufacturing facility in Braunschweig, West Germany.
1982 VIC 20 sales approach 1,000,000 worldwide. The company launches the Commodore 64. Commodore constructs five-inch wafer facility at Costa Meas, California, semiconductor plant.
1983 Commodore 64 monthly sales exceed those of Apple II. More than 4,000 software programs are created by Commodore and independent software companies for use on the machine. Production facilities established at West Chester, Pennsylvania, and Corby, England.

1984 Company launches Plus/4, which is dubbed the "productivity machine" because it has four resident programs: word processing, spreadsheet, data base management, and graphics. Commodore 16 joins company's product mix as exceptionally low-priced, entry-level computer. Irving Gould, chairman, appoints Marshall Smith president, following the departure of Jack Tramiel. Company acquires Amiga Corp., a Santa Clara, California, developer of advanced graphics and sound chip sets.
1985 Commodore announces the Commodore 128 Personal Computer and LCD Portable Computer at the Winter Consumer Electronics Show. Accompanying the 128 Personal Computer are a range of peripherals including the fast disk drive—(C1571); and RGBI/composite color monitor—(C1902); dot matrix printers—(MPS 802 and 803); and other peripheral devices. Company's total worldwide sales of personal computers approach 5,000,000 unit mark. Commodore introduces Amiga Personal Computer. Reviewers hail machine as a technological breakthrough.

DANIEL JANAL

COMMON LISP ON MICROCOMPUTERS

INTRODUCTION

LISP is the second oldest mainstream, high-level programming language (FORTRAN is the oldest). LISP is short for *list processing*. It is widely used in artificial intelligence (AI) programming, along with other symbolic manipulation languages such as PROLOG. LISP and PROLOG are widely used in the development and delivery of expert systems.

LISP has been available on microcomputers since the early days of 8-bit CP/M systems and can be found on just about any system from microcomputers to mainframes. The list-processing features of LISP are just the tip of the iceberg in terms of features and possibilities.

LISP has been known for its numerous dialects, each with its own features and drawbacks. COMMON LISP is a dialect of LISP, which is being pushed as one of the standards within the LISP community. PORTABLE STANDARD LISP and SCHEME are two other, less supported standards. COMMON LISP embodies many of the features found in the major LISP implementations, including MACLISP, INTERLISP and LISP MACHINE LISP. Full implementations of COMMON LISP tend to be large and powerful because of the scope of the COMMON LISP definition.

Many dialects of LISP, including implementations of COMMON LISP, exist on microcomputers. This article will address some of the features of LISP, along with the limitations placed upon a LISP system running on a microcomputer.

HISTORY

LISP was first implemented at MIT by John McCarthy. The most well-known version of this implementation is LISP 1.5. It was originally a batch-oriented system designed for research in AI topics. LISP is based upon work done by Alonzo Chuch in a branch of mathematics called lambda calculus. Symbol manipulation is important to LISP, and symbols and other data items are normally found within a list of some sort; hence, the name *list processing* (LISP).

LISP 1.5 employed dynamically scoped variables like those found in APL, another specialized language. This differs from the lexical scoping used by more conventional programming languages of the time such as ALGOL and FORTRAN.

As LISP grew more popular, it developed into a number of different dialects. One branch, following the dynamically scoped variable approach, includes dialects such as MACLISP, INTERLISP, and FRANZ LISP. Another group started using the lexically scoped variable and included dialects such as T, NIL, and SCHEME. Some dialects actually allowed both forms to be used at the same time, but only one form was the default.

A new dialect, called COMMON LISP, has become important recently. It combines many of the features inherent in previous LISP implementations and is becoming the standard in the LISP community. COMMON LISP supports many of the features found on the more powerful LISP machines, such as vectors with headers, rational numbers, and functional closures.

Microcomputer LISP implementations vary widely in their performance, completeness, and support. The simpler dialects of LISP, such as SCHEME, can utilize smaller environments more effectively. COMMON LISP is much larger and is very comprehensive. It tends to be fully implemented on the more powerful microprocessors such as the 80286, 80386, and 68000, which have a large address space necessary for large applications. Subsets of COMMON LISP are found on processors such as the 8086.

Usually, some form of LISP can be found on just about any microcomputer. The scope of the implementations tend to be based on the popularity of the computer and its capability.

WHAT IS LISP?

LISP is a symbolic processing language and, as such, supports symbols as one of its data types. Symbols have at least two things associated with them: a name that is usually printable and a property list. Like any list, a property list can contain anything: a symbol, another list, a number, and so forth. LISP provides functions to manipulate lists in general and to access a symbol's property list. LISP also supports lists because it is a list-processing language. A list of symbols is displayed as (FIRST SECOND THIRD LAST). The parenthesis marks the beginning and the end of a list.

List elements can be of any type, including symbols and numbers. The elements can also be structures, such as arrays and lists. Lists are accessed sequentially. The CAR function, whose name was obtained from the register on one of the first machines used to implement LISP, is used to access the first element of a list. The CDR function, also named after a register, accesses the rest of the list. A call to the CDR and CAR functions is required to access the second element of a list. CAR and CDR are often called FIRST and NEXT or HEAD and TAIL, which are more mnemonic.

This approach is less efficient than arrays if the elements tend to be accessed randomly but works well if elements are examined sequentially. For this reason, most LISP implementations support lists as well as arrays. One advantage of lists is the ability to share the TAIL of another list. This occurs because LISP stores each element of a list in a CONS cell.

A CONS cell consists of two logical pointers, one to the element and one to the next cell in the list. This type of construction allows lists to share portions of other lists and to dynamically add items to the front of a list. A simple CONS cell referencing two numbers would be written as (1 . 2). Note the spaces around the period.

This differs from the list (1 2), which actually consists of two CONS cells and could also be written as (1 . (2 . NIL)).

NIL is also called the empty list. The list notation is the more compact and conventional way of showing CONS cells and lists. A list that does not end with NIL is normally written as (1 2 . 3). NIL is identical to (), which is an alternate representation. The latter is often found in function definitions with no parameters. This makes the following list representations identical:

```
(1 2)
(1 2 . NIL)
(1 2 . ())
(1 . (2 . NIL))
(1 . (2 . ()))
```

LISP also allows creation of "circular lists," using operations that replace the contents of an existing CONS cell with a reference to a CONS cell in a list that contains the CONS cell being modified. Circular lists are useful data structures for certain types of queues or logically infinite lists.

Initial personel computer (PC)-based LISP implementations used 16-bit pointers in the CONS cells, which allowed access to 64K CONS cells (256Kb), in theory. This is more than many of the minis and mainframes used when LISP first came into being but is much less than existing LISP machines and new implementations of LISP on PCs. CONS cell pointers of 24 and 32 bits are now very common, which allow access to megabytes of memory. This improvement is very important because LISP applications tend to grow very large.

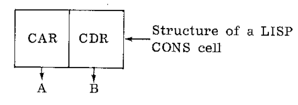
Most LISP implementations support both lists and vectors (or arrays). Although the two are logically similar, there are major differences due to implementation details. Both lists and vectors can contain any data type, but the length of a list can vary, whereas a vector's is usually fixed. Accessing a specific element differs in terms of time because a vector can find the element by indexing, which takes constant time, whereas a list must be searched from the starting point until the element is found. As it turns out, both approaches are valuable, and the appropriate type depends upon the algorithm being used. In general, vectors tend to be more space efficient.

LISP implementations normally include the normal data types found in conventional programming languages, such as integers, floating-point numbers, strings, and files. Structures are also normal fare, and more esoteric data types are also usually found in LISP. Esoteric types include objects for object programming, closures, processes, continuations, stack groups, streams, variable-length integers, rational numbers, and so on.

One of the more major features of LISP is the use of lists to represent functions. The syntax for a function invocation is a list whose first element is a reference to a function, such as the symbol that has been defined to be a function, followed by the list of parameters. This prefix notation is used consistently throughout LISP. The following are examples of LISP expressions, also called symbolic expressions or S expressions:

```
(+ 1 2)
(= A B)
(SIN X)
(+ (* A (SQR X)) (* B X) C)
```


Example 1: (A . B)



Example 2: (A (B . C) D)

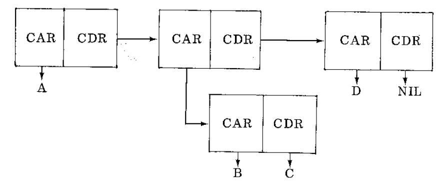


FIGURE 1 List example.

This prefix notation represents the following expressions written in infix notation normally used by conventional programming languages such as BASIC and C.

```
1 + 2
A = B
SIN (X)
(A * SQR(X)) + (B * X) + C
```

LISP prefix notation uses more parenthesis than infix notation because of operator precedence and the requirement that all expressions be lists. In actuality, LISP uses the same amount of punctuation as other languages, but it is more consistent. There is also no ambiguity when using the LISP syntax because there is no operator precedence.

Although dialects of LISP differ in function-definition conventions, the following is representative of how to define a new named function: (DEFINE (ADD X Y) (+ X Y)). Here we have lists within lists. The function being performed is DEFINE. The function being defined is ADD, and it has two parameters X and Y. The last list is the operation performed when ADD is referenced. In this case, the function is "+," which is usually the built-in function for adding two numbers together. This function invocation uses the values of the parameters X and Y.

LISP is based on lambda calculus, which often shows up directly within LISP. A lambda expression describes a method of substitution that can be used to explain how a function operates. The previous example can also be described, setting the functional value of ADD to (LAMBDA (X Y) (+ X Y)). Here LAMBDA is not a function but, rather, a function-definition indicator.

It essentially defines an unnamed function, which can be used as the function referenced by the first item in a list that is a function invocation. In some LISP implementations, the previous function definition example is equivalent to (SET! ADD (LAMBDA (X Y) (+ X Y)), where the SET! function, defined in the SCHEME dialect of LISP, assigns the value of the LAMBDA expression to the variable ADD. Evaluation of a variable, as you may have quessed, results in the value assigned to the variable. Common LISP uses SETQ instead of SET!; only the name is changed.

In most LISP implementations, but not all, it is possible to use a LAMBDA expression as the first element of a list. In this case, the LAMBDA expression will process the parameters in the rest of the list. For example, ((LAMBDA (X Y) (— Y X)) 1 2).

Although the LAMBDA expression in this case is simple, it does show the syntax and power or LISP. LAMBDA expressions can also be used as parameters. This is similar to procedural parameters in Pascal and pointers to functions in C. The following is a LISP example:

```
((LAMBDA (X Y F) (F X Y))
1
2
(LAMBDA (X Y) (-Y X)))
```

LAMBDA expressions have their parameters evaluated when the LAMBDA expression is used. However, it is possible to "quote" a parameter so that the parameter is not evaluated. Quoting is done using the built-in function QUOTE, as follows:

```
(SOME_FUNCTION (QUOTE (1 2 3)) 4 5)
(SOME_FUNCTION '(1 2 3) 4 5)
```

QUOTE is the first element of the list that is the first parameter because it is a function that is evaluated. However, QUOTE does evaluate its parameters and is often called an FLAMBDA expression, which works like a LAMBDA expression but does not evaluate its parameters. Some implementations allow FLAMBDA expressions, whereas others only supply QUOTE. The second expression in the last example shows the conventional shorthand method of indicating QUOTE.

LAMBDA functions can handle a specific number of parameters, as defined previously. LISP functions can also handle parameter lists that vary, but the convention varies according to LISP dialect. Common LISP uses key words like &OPTIONAL, &REST, &KEY, and &AUX to specify this syntax. The key words are followed by the names of the parameters. &OPTIONAL parameters may appear in the parameter list when the function is called and are bound in the order they are defined. &REST is bound to the list of any remaining parameters that are not bound to the main or optional parameters. Only one variable name can appear after &REST. A run-time error occurs if &REST is not included in a function definition, and the function is called so that there are parameters that cannot be bound to the main or optional parameters. &KEY indicates that parameters are passed by a key word. The key word is specified using a colon prefix to it, followed by the value of the key word parameter. The order of key word

parameter pairs is not important as they are removed from the list of parameters before the main, optional, and &REST parameters are bound. &AUX indicates that the names following it are local variables.

Including an argument name in a list indicates that the parameter should be initialized to a particular value instead of being unbound, which is a detectable state in most LISP implementations. The value of the argument is the initialization value if there is no corresponding parameter for the argument. &KEY parameters are slightly different in that a key word, as in ((:NEW-KEYWORD K3) value. &AUX variables are not bound with parameters, so they will only have an initial value if they are assigned using this technique.

The following is a sample function definition incorporating these parameter definition key words.

```
(DEFINE SAMPLE (P1 P2 &OPTIONAL 01 (02 2)
                      &REST R
                      &KEY K1 (K2 2) ((:KEY3 K3) 3)
                      &AUX L1 L2)
               (LIST P1 P2 02 R K2 K3))

SAMPLE could be called using any of the following:
   (SAMPLE 1 2)                  (1 2 2 () 2 3)
   (SAMPLE 1 2 3 4)              (1 2 4 () 2 3)
   (SAMPLE 1 2 3 4 5 6)          (1 2 4 (5 6) 2 3)
   (SAMPLE 1 2 :K1 3 :K3 4)      (1 2 2 () 3 4)
```

The variables 01 and K1 are unbound unless a third parameter of :K1 appears in the parameter list, respectively. Key words appear before their respective value and are not order dependent. The key word &ALLOW-OTHER-KEYS can be used to allow the &REST variable retain any keys not specifically indicated in the LAMBDA parameter definitions.

The entire parameter list can be placed into a single variable using the &WHOLE key word followed by the variable name. Other key words should not be included, except &AUX. &WHOLE tends to be used with MACROs, discussed later, so that the original function invocation can be altered. &WHOLE differs from &REST in that the argument is bound to the list that contains the function invocation, whereas &REST is the list of the remaining parameter values.

SCHEME uses the parameter list syntax to specify a fixed number of parameters or COMMON LISP—style key words like &WHOLE and &REST. SCHEME argument lists have three general formats:

SCHEME	COMMON LISP
R	(&REST R)
(P1 P2)	(P1 P2)
(P1 P2 . R)	(P1 P2 &REST R)

SCHEME uses a pattern-matching method of parameter-argument binding but lacks &OPTIONAL and &KEY support. &AUX parameters can be included in SCHEME, using the LET construct.

The use of lists to describe functions is exploited in LISP by the use of LISP MACROs. A MACRO is logically equivalent to MACROS in assembler and C, except that LISP MACROs convert a list structure to another list

structure, which is then used as the expression to be evaluated. In fact, LISP MACROs are simply LISP functions that manipulate lists so that LISP provides all of its facilities when using a MACRO. C and assembler do not provide this type of capability. Using MACROs is very simple because extracting items from a list and building up a new list is something for which LISP is made.

The implementation of MACROs is one area where LISP implementations tend to differ. The differences tend to occur when the transformation is performed and how the parameters are passed to the MACRO. Some LISP implementations perform MACRO expansion when a function is defined, whereas others perform it when an S expression is evaluated. In general, a MACRO receives the list that is being evaluated as a set of parameters and returns a list that is evaluated again. This, in turn, can be another MACRO invocation if the first item in the list is a MACRO.

Systems that expend MACROs during evaluation have two options: (*a*) to perform the expansion every time; (*b*) to perform the expansion the first time and then modify the list used for the original invocation with the expanded version. The latter is faster because the expansion of the MACRO can be time consuming, but the former allows the MACRO invocation form to be retained. This is important in systems that allow function definitions to be accessed for display and edit purposes.

A combined form is often used in more sophisticated systems, which retain both the original and expended form of the MACRO. The original list is modified such that a special MACRO invocation function is used that takes the original and expanded forms as parameters. The function always evaluates the expanded version and ignores the original. The advantage is that the expansion is done once and the original form of the MACRO is retained. The disadvantage is that there is an additional function call associated with each expanded MACRO. The latter tends to be a minimal overhead. Systems that include this feature normally have special printing functions that can display a function definition so that these combined forms are automatically reduced back to their original form.

The following is a sample MACRO definition:

```
(MACRO (TEST-MACRO F A B)
(LIST (LAMBDA (X) (F A X A)) B))
```

This MACRO does not evaluate its parameters but creates a list, which is an S expression that will be evaluated in place of the MACRO invocation. An evaluation of (TEST-MACRO LIST 1 2) would cause the MACRO to return ((LAMBDA (X) (LIST 1 X 1)) 2) which, in turn, evaluates to the list (1 2 1). Another type of MACRO often found in LISP systems is READMACRO. This differs significantly from the more conventional MACRO just discussed. In fact, READMACROs are rather like string processors. Essentially, a READMACRO is a special character that is recognized by the LISP input function. The input function then invokes the corresponding function that returns the next item in the input stream that may be processed in some fashion. For example, the shorthand for the QUOTE function is implemented as a READMACRO, where the quote character invokes a function that reads the next item in the input string. The object read is of the form (QUOTE object).

Although both types of MACROs perform a conversion function, the conventional MACRO is the one found most often within a program. READMACRO tends to be used when building a special input syntax; however, MACRO tends to use various READMACROs extensively to simplify the creation of the list that is returned by a MACRO. One reason for this use of READMACRO is that MACRO tends to perform a good deal of list construction. However, the MACRO definition can become confusing because the contents of the list are interspersed with the list construction part of a function. The following example may help to clarify this situation: '(These items are quoted ,@(LIST A B) ,(LIST A B) (LIST A B)).

READMACROs are started with a special character like "and." The latter is called a backquote. Backquote indicates that the subsequent S expression should be quoted but that certain elements should be evaluated. This differs from the normal quote function, which evaluates none if its parameters. The comma is used as a special character within a backquote definition. It indicates that the subsequent item is to be evaluated and its result inserted into the MACRO result list. A single comma indicates that the result must be a list that is appended to the list being created. A comma followed by an at sign (@) places the result as an element in the list. The result of the previous example would be (assume A is 2 and B is 3):

```
(APPEND (QUOTE (These items are quoted))
        (LIST (LIST A B))
        (LIST A B)
        (QUOTE (LIST A B)))
```

The evaluated form results in the following: (These items are quoted (2 3) 2 3 (LIST A B)).

Note how closely the evaluated form matches the original form, whereas the intermediate form contains superfluous but necessary functions like APPEND, QUOTE, and LIST.

READMACROs can be used anywhere and do not have to be restricted to MACROs. Often, many data types are entered using READMACROs. For example, arrays are often indicated using conventions like square brackets, as in [1 2 3], or a list that is prefixed by a pound sign, as in #(1 2 3). These can be implemented by making the left square bracket or pound sign a READMACRO character that reads the subsequent items until the matching character is found. The items read are then saved in an array.

Unlike MACROs, READMACROs are always invoked when something is read using the STANDARD LISP READ function. READMACROs cannot be expanded after something is read in.

LISP, by nature, is a functional language, versus a statement-oriented language such as BASIC or Pascal. LISP only has functions, and each function returns a value. Sometimes this value is undefined or ignored. The consistency benefits the language by simplifying it.

LISP contains most control structures found in other languages, including loops, conditionals, and case statements. Unfortunately, this is one area where different dialects of LISP diverge. However, most implementations support the LISP 1.5 form of the conditional statement COND, which has the form:

```
(COND (test_1 result_1)
      (test_2 result_2)
      (test_3 result_3))
```

The "result" of this expression is the first one whose "test" value is not false. Variations allow multiple expressions to occur in the condition lists where the result is the last expression evaluated.

Most LISP implementations also include the more familiar IF expression in addition to COND. For example, (IF test result_true result_false). The IF function has only one condition, and the false result can often be omitted, in which case it defaults to NIL. IF is easier to use than COND in most cases because testing is often restricted to a single condition. IF can be implemented as a MACRO, which expands into COND.

LISP implements a block structure, using constructs such as LET and LETREC. Like most things in LISP, these functions return a single result. The form for these two functions is identical and is described in the following example:

```
(LET (local_1 local_2 local_3)
     exp_1
     exp_2
     exp_3)
```

There can be any number of local variables (i.e., local_1) and expressions (exp_1). The result of the LET function is the last expression. Local variables are names like X or APPLE. The variables can be initialized by placing the name and its value in a list where the initial value will be evaluated.

```
(LET (V1
      (V2 2)
      (V3 (LAMBDA (X) (+ X 3))
     )
     (SETQ V1 1)
     (LIST V1 V2 V3)
)
```

Note that SETQ or SET! in some dialects like SCHEME, is like an assignment statement in languages like BASIC and C. LET is normally implemented via a MACRO that would convert the previous example into

```
((LAMBDA (V1 V2 V3)
         (SETQ V1 1)
         (LIST V1 V2 V3)
)
UNBOUND 2 (LAMBDA (X) (+ X 3))
)
```

LETREC is similar to LET, except that initial values of variables can refer to other local variables. There are other LISP functions similar to LET, which perform other useful functions, and these can easily be expanded using MACROs.

It is interesting to note that these functions can be used anywhere, even as a parameter to another function. For example,

```
(LIST (LET ((X (SIN 45))
            (Y (SIN 30))
           )
           (+ (SQR X) (SQR Y))
      )
      45
      (LET ((A (LIST 1 2 3)
            (B (CONS 1 2)
           )
           (APPEND A A B)
      )
      '(This is an example)
)
```

Although the example is contrived, it shows how LET can be easily incorporated into a parameter to another expression. It also allows local variables to be included within function definitions in dialects of LISP that do not support the COMMON LISP &AUX feature.

Most implementations also support CATCH and THROW, which are similar to C's SETJMP and LONGJMP functions. THROW allows a program to return directly back to a preceding CATCH. However, CATCH and THROW are more powerful and controllable. For example, another LISP function, UNWIND_PROTECT, can be called after a CATCH but before THROW is called. The second function associated with UNWIND_PROTECT will always be executed. This cleanup function allows files or data structures to be cleaned up regardless of what happens. CATCH and THROW are used to implement error handling, as well as general control features. The syntax for these functions is

```
(CATCH tag expression)
(THROW tag expression)
(UNWIND_PROTECT expression exit-expression)
```

The expression in CATCH is evaluated, and its result is returned as the result of the CATCH expression, if all goes normally. However, CATCH returns the expression result of the THROW function with a matching TAG if the THROW function has been involked through the CATCH expression. All intervening function invocations are thrown away. This is a very efficient way of returning a value from a function without having to perform tests along the way to see if it is time to exit. The THROW function can also be used to handle error notification. The process of finding the matching CATCH function and cleaning up the stack is called unwinding.

UNWIND-PROTECT is used to handle the unwinding in a controlled fashion. It evaluates its expression and always evaluates the exit-expression, even if a THROW function sends control to a CATCH expression, which invokes both the UNWIND-PROTECT and the THROW function. The result exit-expression is evaluated when the UNWIND-PROTECT expression returns a result after it is determined that a THROW through the UNWIND-PROTECT is to occur. The result of an UNWIND-PROTECT function is the result of its expression if a THROW does not occur.

A CATCH/THROW tag is normally a symbol. Implementations often allow NIL to act as a CATCH-all. Unfortunately, it is not usually possible to determine what the CATCH tag was at that point.

Continuations are a more general form of CATCH and THROW. Unfortunately, continuations are only implemented in a limited number of LISP dialects, such as PC SCHEME. COMMON LISP does not support continuations. Continuations also allow creation of coroutines, a useful process found in only a few other languages such as MODULA-2. However, continuations are actually more general and can be used to implement backtracking algorithms like those found in PROLOG.

Continuations are normally implemented using the CALL-WITH-CURRENT-CONTINUATION function, which is a function of one parameter that should be passed a continuation as its only parameter. The continuation is, in turn, a function that also takes a parameter. Calling a continuation causes execution to continue as if the CALL-WITH-CURRENT-CONTINUATION returns the parameter passed to the continuation. CALL/CC is used as an abbreviation for CALL-WITH-CURRENT-CONTINUATION. For example,

```
(CALL/CC (LAMBDA (C) (+ 2 3)))
(CALL/CC (LAMBDA (C) (+ 2 (C 3))))
```

The result of the first example is 5, which is the result of the LAMBDA expression. The continuation parameter, C, is ignored. The result of the second example is 3, and the "+" and LAMBDA functions never get a chance to return a value. This operation is almost identical to a CATCH and THROW pair. The difference is that CATCH and THROW use tags to indicate matching items, and CALL/CC is used in place of CATCH. The continuation is identical to THROW. UNWIND-PROTECT can be implemented by creating a new continuation and passing it on to the expression invoked by UNWIND-PROTECT. The new UNWIND-PROTECT function would have to determine how the expression is terminated and either return or invoke the continuation passed to it.

Continuations are more general than CATCH and THROW. CALL/CC can be used to implement coroutines, nonpreemptive multitasking, and backtracking algorithms because a continuation can be invoked more than once. This allows a continuation to be invoked with different parameters.

Some PC-based LISP implementations also support objects such as stack groups and engines, which are a form of multitasking. The advantage of including these features within LISP is that the various processes can share LISP data. Otherwise, they would be limited to exchanging numeric data and strings because most languages would be unable to process items such as lists and rational numbers.

The scope of variables within LISP is another area where differences appear among dialects. Dynamically scoped variables were very common in previous LISP implementations. Essentially, a reference to a dynamic variable is a reference to the latest definition of a variable using the same symbolic name. A new definition within an active function is all that is needed to temporarily use a new value. The previous value is used when the function exits. For example, PROMPT could be a variable that contains a string used for a prompt to a question within a function. The system default value of PROMPT is a question mark (?). A new function, which calls the prompting function, is defined with the variable PROMPT as a parameter or local variable. The value of this PROMPT variable will be

OK (Y/N)?. Calling the PROMPT function directly uses "?" as the prompt, whereas calling the new function causes the prompt to be OK (Y/N)?. In some dialects of LISP, dynamic variables are indicated by a functional form like (FLUID PROMPT). APL is another language that support dynamically scoped variables.

Lexically scoped variables are the alternative to dynamically scoped variables. Lexical scoping is used in languages such as ALGOL, C, and Pascal. References to variables must be available when a function is defined that means the variable must be a global variable or a parameter or local variable of a function that contains the variable reference.

Lexical scoping tends to be more efficiently implemented on PCs and allows easier and more efficient compilation of LISP. COMMON LISP and SCHEME are two dialects that use lexical scoping as the preferred scoping method. Both allow dynamic variables but require explicit indicators, such as SPECIAL or FLUID, to show that a variable is not lexically scoped.

Lexical variables tend to be allocated in a stacklike fashion where they can be indexed by offsets into the stack. They are used in the same fashion as variables defined in Pascal or C, with regard to scope. Allocation and deallocation are preformed by adjusting the stack pointer.

Dynamic variables, on the other hand, require a different approach because references are to the most recent incarnation of a symbol. Allocation, deallocation, and referencing variables are more difficult because of the properties of dynamic variables.

The two alternatives for implementing dynamic variables are "shallow" and "deep" binding. Shallow binding uses the property list of a symbol to store the list of references to previous values, and the value of the symbol is its most recent value. Each time a symbol is used as a parameter or local variable, the current value is saved on the property list and a new value is assigned. The saved value is removed from the property list and reassigned to the variable when the function defining the symbol is exited. The advantage to this approach is that all references to a variable are to the symbol, which is a very quick operation. The disadvantage is the storing and restoring of previous values.

Deep binding works in a different fashion. In this case, there is a list of bindings with symbol and value pairs. The value of a variable is found by searching the list until the matching symbol is found. The ordering of the dynamic variables is maintained by the order of the list because the latest incarnations are at the front of the list. Returning from a function is very quick because the variable bindings are updated by simply changing the pointer to the front of the list to a prior position.

Shallow binding works well for single tasking, dynamically scoped implementations but is inappropriate for multitasking or lexically scoped implementations. Lexically scoped implementations access local variables and parameters by using stack offsets. Multitasking implementations, including those that support continuations, must use deep binding because each execution thread contains its own view of the world. It would be invalid to have one function undo the work of another using shallow binding. Actually, the speed penalty for accessing variables in a lexically scoped environment is very small because most variables are on the evaluation stack and do not use the dynamic variables.

Tail recursion optimization is implemented in many LISP interpreters and compilers, although it appears more often in implementations that use lexically bound variables. The optimization consists of converting the last

call in a function definition to a jump. This approach does not consume stack space as a normal function call would. In fact, normal iteration functions can be easily described using optimized function calls. For example,

```
(DEFINE  (SAMPLE-1 X)
         (PLUS X 1))
(DEFINE  (SAMPLE-2 X)
         (IF (LISTP X) (SAMPLE-2 (CDR X)) (CONS X 1))
```

Sample-1 is tail recursive, optimizing the call to PLUS. Sample-2 is more complex in that the IF function is not optimized but, rather, the second reference to SAMPLE-2 and CONS are because one of the two will be the last function executed by IF. In many cases, the code generated by this optimization is more efficient than loop constructs.

LISP was initially implemented as a batch mode interpreter. It has since evolved to an interactive interpreter/compiler environment with some of the most advanced development tools available. In fact, LISP machines tend to be state of the art in both hardware and software support. PC-based implementations tend to be less sophisticated but retain the interactive nature normally associated with LISP.

GARBAGE COLLECTION

LISP allocates new objects such as lists and arrays from a free memory pool. CONS is a built-in function that allocates a new list cell, but there is no function that explicitly deallocates a list cell or any other object, for that matter. Instead, an object is considered reusable when there are no longer any references to it. Such objects that must be periodically collected for reuse are considered garbage. Garbage collection is the process of recovering unused objects.

There are various methods of performing garbage collection and the method used is important to the overall performance of the system because it must be done periodically. The simplest method is the two-pass mark–sweep process, which is done when all free memory has been allocated. The first pass scans all the data links in memory and marks each item that is not being used. The second pass collects the newly available objects. Although this approach is relatively quick, it may still require a few seconds to complete the garbage collection operation. Luckily, garbage collection does not have to be performed often.

Compaction is another aspect of some garbage collection algorithms. All objects in use are moved to one end of the memory pool, leaving the other end free. This takes additional time but makes object allocation faster and allows the allocation of objects that are very large. Otherwise, free space can become fragmented, making the allocation of large objects difficult, if not impossible. Some LISP machines use a concurrent garbage collection algorithm that is not normally implemented on PCs. Garbage collection is performed while a program is running, so a program should never have to stop and perform a garbage collection. This approach slows down the overall operation of an application but the average operating time is about the same. It also prevents any pauses, which are much more noticeable.

As you may have guessed, there is no such thing as a free lunch in LISP. The benefits of garbage collection normally outweigh the performance penalty.

LISP ON MICROCOMPUTER ENVIRONMENTS

LISP has been implemented on a number of microcomputer environments, including hardware based on the Zilog Z80, Intel 8080/8088/8086/80286/80386, and the Motorola 68000. Various implementations can be found running under CP/M, PC-DOS, MS-DOS, XENIX, GEM, and so on. Some take advantage of the machine and operating system, whereas others simply provide basic LISP text-based support. The primary limitation on 8-bit machines is memory space. Sixty-four kilobytes, shared among an interpreter and operating system, does not leave much for a LISP application. These implementations are limited to very small prototype tools or as learning aids. The 8-bit versions are almost all interpreter-based systems.

The 16-bit environments fare much better. Memory limitations are normally above 512Kb, which allows allocation of at least 64K CONS cells, this is sufficient for many applications, including expert systems. The 80286, which has an extended memory limit of 16Mb, has at least one entry that allows access to extended memory, thereby raising the maximum number of CONS cells.

The largest group of LISP implementations runs under PC-DOS and its cousin MS-DOS on 8088/8086/80286-based PCs. The implementations range from simple low-level implementations to the powerful, partial implementation of Common LISP. Many include resident editors written in LISP. The more complete LISP implementations also support either a pseudocode (Pcode) or native–code compiler. The difference is typically a trade-off between size—smaller for the Pcode version—and speed—the native–code versions are faster.

The additional functionality of the 16-bit implementations is noticeable in other areas too. On-line help and documentation are found in the more expensive products. Debugging and development tools are also more mature, although they tend to be less powerful than those found on LISP machines.

Overall, the 16-bit implementations now provide full-featured systems for real application development and delivery on PCs. Access to large amounts of memory is often provided using the 80286 in protected mode.

The 32-bit microcomputers, such as the 68000, have been used on workstations for a number of years. The LISP implementations on these machines rival those of the LISP machines in terms of speed and power, although the latter still holds an edge. However, the 32-bit microcomputers are showing up in some low-end machines such as the Atari ST and Commodore Amiga. Although these machines were initially short on software, the amount and quality has increased, including LISP implementations.

The Intel 80xxx family of microprocessors is a popular base on which to build LISP. The 8088/8086 microprocessors are limited to 1Mb of memory, as are the 80286 and 80386 when run in native mode, which is compatible with the 8088/8086. The protected mode of the 80286 provides access to 16Mb of memory. These microprocessors use a 16-bit architecture, whereas

the 80386 is based on a 32-bit architecture. The 80386, running in protected mode, provides access to gigabytes of memory directly, has an upper logical limit of terabytes, and has a virtual memory system to fit most needs. Of course, proper software is needed to exploit these features.

The 16- and 32-bit microcomputers provide a reasonable platform for LISP-based systems. New implementations will only expend upon an already increasing base.

EXISTING IMPLEMENTATIONS

The number of LISP implementations on microcomputers is relatively large and may rival the number of C compilers on the market. A variety of dialects are also supported, including some unique to the microcomputer world.

Two of the oldest PC_based LISP systems are MULISP and TLC LISP, both of which have versions that run on 8-bit CP/M-based microcomputers. These systems also run on 16-bit MS-DOS and CP/M-86-based micros. The difference between the 8- and 16-bit versions is significant, including the increased memory utilization. The number of built-in functions and overall support are also greatly expended. Optional native code compilers are available for the 16-bit versions.

Integral Quality is one company with a Common LISP product called IQ COMMON LISP. In addition to supporting much of the COMMON LISP definition, it also supports many of the functions found on the IBM PC, including graphics and access to MS-DOS. It is a 16-bit implementation that runs on the 8088/8086 and 80286 in native mode. IQ LISP is also available, but it is not as compatible with Common LISP as IQ COMMON LISP but, rather, with INTERLISP, a dialect of LISP supported by Xerox. Native code compilers are also available. IQ LISP implementations are window based and include a very good debugging environment.

Gold Hill has a number of versions of Common LISP from a strictly 16-bit version for the 8088/8086 and the 80286 in native mode to a full-fledged version running in protected mode on the 80286 and 80386. Gold Hill has one of the more complete lines of Common LISP for the Intel microprocessor family. Compilers are available for the protected-mode versions, and the compilers can generate programs for systems running in native mode, although the accessible memory is limited by the microprocessor where the programs are to be run. Various tools, including an EMACS-style screen editor, are included. An interactive tutorial is supplied along with Winston's book on LISP. Gold Hill has extended its implementation of COMMON LISP to include features found in LISP MACHINE LISP. Gold Hill also supplies COMMON LISP for the Intel hypercube, multiprocessor product. This version supports a library of interprocess communication functions.

WALTZ LISP is an implementation of FRANZ LISP, which is another popular dialect of LISP. It runs on 8-bit microprocessors and 16-bit Intel microprocessors and is a dynamically scoped LISP that comes with an implementation of PROLOG written in LISP. The system is useful in learning about LISP, especially FRANZ LISP, but differs significantly from Common LISP.

PC SCHEME, from Texas Instruments (TI), is a version of SCHEME that supports windows, continuations, and engines. It is one of the few microcomputer-based LISP implementations that supports multitasking.

PC SCHEME use of expanded or extended memory for large programs and data. PC SCHEME uses an incremental compiler, which provides interactive development with the execution speed of compiled code. Support for object-oriented programming is supplied with the SCOOPS package. PC SCHEME also includes an EMACS-style editor written in SCHEME, although the source code is not included. The system is used as the basis for two TI products called Personal Consultant Plus and Personal Consultant Easy, which are expert system development tools. The former includes direct access to PC SCHEME for enhancement of expert systems.

The LISP Company provides TLC LISP, which is based on MACLISP but has been enhanced in many areas, including support for an object-oriented programming environment. It was one of the first implementations to support tail recursion optimization and functions as first-class objects. The system supports an assembler, compiler, and screen editor written in LISP.

XLISP, by David Betz, is a public domain version of LISP, written in C. The source code is available from a number of sources, including user groups and bulletin boards. It was originally implemented to evaluate object-oriented programming and, as such, includes supports for objects and message dispatching. XLISP has been run on systems ranging from 8-bit microcomputers to superminis. All that is really necessary is to have a good C compiler and a working knowledge of C and LISP.

The 68000 microprocessor is the heart of a number of powerful workstations that support LISP, along with a host of other languages and applications. The 68000 is also used in three major low-cost systems from Apple, Commodore, and Atari. LISP implementations exist for these computers, but they tend to be less sophisticated than their workstation counterparts. Even so, these implementations are on a par with those found running under MS-DOS. They also tend to take advantage of the more sophisticated graphics interface found on these machines.

The major advantage of the 68000-based LISP implementations is the 32-bit internal architecture of the 68000, which handles 32-bit address with ease. CONS cells for large LISP implementations often consist of a pair of 32-bit addresses. The 8086-based implementations often require multiple instructions to manipulate addresses of this size or resort to special encoding schemes to reduce the size of a CONS cell.

Metacomco has a version of Cambridge LISP, which runs on the Apple Macintosh, the Atari ST series, and the Commodore Amiga. This 68000-based implementation consists of an interpreter and a compiler. It utilizes 32-bit addresses in its CONS cells for access to all available memory. The dialect is Cambridge LISP, which is a dynamically scoped LISP implementation. It does provide access to the various graphic and system functions on the system on which it is running.

Other 68000-based LISP implementations include ExperIntelligence's EXPERLISP and XLISP, which was covered earlier in this section. Implementation on 68000-based workstations is not addressed here, as it tends to be on the same order as LISP machine implementations, which include extensive support for COMMON LISP, windows, and operating environments. The environments tend to be more advanced than anything found on the lower-cost microcomputer systems on the market.

LISP implementations on microcomputers are varied and range from small, experimental tools to large, full-featured development compilers. Only a few of the major implementations are mentioned in this section. New

implementations are appearing, and existing ones are being enhanced. The common thread of LISP permeates them all, even though the incompatibilities may be varied. There are even differences between COMMON LISP implementations because of the various enhancements and because most implementations do not include the complete COMMON LISP definition, which is substantial.

The main failing of most microcomputer-based LISP implementations is in the area of file and record support. This tends to be due to historical compatibility and the original use of most systems as learning or AI-related tools. The limited file and record support made implementations of more conventional applications difficult at best. Improvements are being made in this area, but LISP standards tend to be deficient here.

For the most part, microcomputer-based LISP implementations are quality products, and there is a wide variety of implementations and dialects available. Most implementations either follow the COMMON LISP standard, SCHEME, or none at all. The latter implementations offer little in terms of transportability.

FUTURE DIRECTIONS

Initial LISP implementations under CP/M and DOS were limited in too many ways. which restircted their usefulness to learning tools. Existing implementations are far superior and practical development environments for large projects, including expert system development. This has come about with the utilization of more memory, support of graphics, and the use of compilers.

Two standards seem to be emerging from the various dialects of LISP. The foremost is COMMON LISP. This specification has broad support within the LISP community, and a number of partial implementations exist. In fact, very few complete implementations exist, even on LISP machines, because of the size of the COMMON LISP specification. Although not all encompassing, COMMON LISP does try to address almost all aspects of LISP and the various dialects. More complete implementations of COMMON LISP will be found on microcomputers as time goes on.

The other standard is SCHEME. SCHEME is smaller and easier to maintain and learn than COMMON LISP. This also makes it easier to optimize because of the reduced number of considerations. It is this simplicity, along with its power, that makes SCHEME attractive to a large number of people.

The three areas of advancement for microcomputer-based LISP systems will be in compilers, graphics, and support functions. Compilers exist now, but there is a limited amount of optimization. LISP can be as efficient as an optimized compiler for a conventional language such as C or FORTRAN, but, like any optimized compiler, it takes a good deal of effort to build and support such a product.

Graphical interfaces on LISP machines have been the dream of many developers, and current microcomputer systems now have the resolution and computing power to support such an interface. Existing microcomputer-based LISP systems do not take advantage of the underlying system to the same degree as the LISP machines, but this is changing. Although the dedicated LISP machines will probably continue to be in the forefront in this area, each will advance the state of the art.

The LISP machine support packages are another area where microcomputers are moving forward. These include functions such as debugging, source code control, graphic design tools, and editors. Most microcomputer-based LISP systems have support for these functions, but to a very limited degree. The lack of support is mainly due to a limitation in performance and main memory capacity. These functions are becoming more powerful as time goes by.

LISP implementations on microcomputers will continue to improve. The improvements will be driven by the requirements of new applications, such as expert systems. LISP systems will continue to require large amounts of memory but will utilize it better than most existing development languages.

SUMMARY

Microcomputer-based LISP systems range from basic learning tools to full-featured development systems. LISP features found on LISP machines and mainframes but not on PCs tend to be multitasking, specialized graphics support, large virtual memory, and specialized development tools. These features are usually not necessary for a large majority of applications, which means that the PCs are very appropriate for using LISP.

LISP is a serious development tool in the same league as BASIC, C, COBOL, FORTRAN, and Pascal. It has certain features that make it capable of symbolic applications, and LISP also works as a general applications language.

BIBLIOGRAPHY

Abelson, H., and G. J. Sussman, with J. Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, MA, 1985.

Allen, J., *Anatomy of Lisp*, McGraw Hill, New York, 1978.

Church, A., *The Calculi of Lambda-Conversion*, Princeton University Press, Princeton, NJ, 1941.

Goldberg, A., and D. Rodson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA, 1983.

Griss, M. L., *Portable Standard Lisp, A Brief Overview*, Utah Symbolic Computation Group, Operating Note 58, University of Utah, Salt Lake City, UT, 1981.

Kowalski, J., P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin, *Lisp 1.5 Programmer's Manual*, 2nd ed., MIT Press, Cambridge, MA, 1965.

Moon, D., *MacLISP Reference Manual*, Technical Report, MIT Laboratory for Computer Science, Cambridge, MA, 1978.

Moon, D., and D. Weinreb, *Lisp Machine Manual*, Technical Report, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1981.

Steele, G. L., Jr., "Debunking the 'Expensive Procedure Call' Myth," *Proc. Natl. Conf. ACM*, 153–162 (1982).

Steele, G. L., Jr., and G. J. Sussman, *Scheme: An Interpreter for the Extended Lambda Calculus*, Memo 349, MIT Artificial Intelligence Laboratory, Cambrideg, MA,

Steele, G. L., Jr., *Common Lisp, The Language*, Digital Press, Burlington, MA, 1984.
Teitelman, W., *Interlisp Reference Manual*, Technical Report, Xerox Palo Alto Research Center, Palo Alto, CA, 1974.
Wand, M., "Continuation-Based Program Transformation Strategies," *J. ACM*, 27(1):164–180 (1981).

WILLIAM G. WONG

COMPILER DESIGN

1. THE PROBLEM

Programming languages are notations for describing algorithms—finite computations operating upon states to produce results. Every programming language definition includes the concept of a state and specifies the kinds of objects that make up states and results. Although the characteristics of states and objects may vary widely among programming languages, it is generally possible to model those of one language by those of another. For example, Pascal allows the user to define objects that are sets and include them in the state. Assembly languages do not allow set objetcs, but the assembly language programmer can model them by sequence of bits.

A compiler is a program that accepts a description of an algorithm in a human-oriented source language and creates an "equivalent" algorithm in a machine-oriented target language. Equivalence is defined in terms of the input–output behavior of the two algorithms [*1*]: If the initial state of the target algorithm is a model of the initial state of the source algorithm, the result produced by the target algorithm is a model of the result produced by the source algorithm. There are usually several target algorithms that are equivalent to a given source algorithm according to this definition, and we would like the computer to produce one that is "good" in some sense. The compiler should also verify (insofar as possible) that the given source algorithm conforms to the rules of the source language.

1.1 Source Language Characteristics

Each source language has a single defining document. This document is often a formal standard and sometimes a "report" issued by the language designer. It is important to base the design on such a document, rather than on some tutorial description of the language, because the latter usually does not distinguish clearly between what is permitted and what is prohibited; tutorial descriptions are informal and sometimes omit parts of the language. A compiler designer must thoroughly understand the properties of the source language being implemented. The purpose of this section is to outline the important characteristics of current programming languages, providing a road map for the study of a language definition (Reference *2* provides a more complete treatment).

The most important characteristics of a programming language are the data objects it provides and the operations defined upon them. If an object can be accessed only in its entirety, we say that it is elementary; objects consisting of a collection of distinct components that may be manipulated individually are composite. Every operation interprets its operands in a specific way, and particular operations may not be applicable to every object. Such restrictions are embodied in the concept of type. A type defines a

subset of the universe of objects according to the operations that may be applied to those objects. In practice, the objects belonging to a particular type are either all elementary or all composite; we therefore apply the terms "elementary" and "composite" to types as well as objects.

The number of objects in an elementary type may be finite (e.g., truth values or characters) or infinite (e.g., integers or reals). They may be defined to be in one-to-one correspondence with the natural numbers, they may have an ordering defined upon them without such a correspondence, or they may be completely unrelated to one another. These properties constrain the choice of target language objects used to model them in an implementation of the source language.

Composite types define selectors used to access the components of a composite object. They may be computable (as in the case of array indices) or noncomputable (as in the case of record field selectors). The size of a composite object may be fixed by its type, fixed at the time the object is declared (so that different objects of the same type may have different sizes), or flexible (so that the size of the object itself may vary). These properties affect the complexity of the target machine state needed to model the source language state and the cost of access to a component of a composite object.

Operations that return objects can be composed into expressions. An expression defines a data flow relation among its components: No operator can be evaluated before its operands have been evaluated. Not all operators require evaluation of all of their operands. For example, a conditional operator requires evaluation of the condition and one of the other operands (the choice depending upon the outcome of the condition). If evaluation of an expression alters the state, we say that it has a side effect. When no component of an expression has a side effect, the components may be evaluated in any order that satisfies the data flow relation without changing the expression's result. If some components have side effects, different evaluation orders may give different results. A language definition may state that only the data flow relation is relevant in determining the order of evaluation within an expression (in which case the programmer must ensure that side effects do not influence the expression's result), or it may pose additional restrictions such as strict left-to-right evaluation.

Two actions that alter the state may be composed in three ways: serial, parallel, or collateral. Serial composition means that the first action must be completed before the second is begun, parallel composition means that the two actions must be carried out simultaneously, and collateral composition means that they may be carried out simultaneously or either may be completed before the other is begun. The source language definition specifies which method is to be used in a particular construct. For example, many languages use the semicolon to indicate sequential composition of two actions. The multiple assignment $a,b:=b,a$, which interchanges the values of two variables, is a parallel composition of two actions. In many languages, the evaluation of the arguments of a procedure is defined to proceed collaterally, thus allowing the implementer to compute their values in any order. Again, the programmer must ensure that collateral composition leads to a well-defined result.

The state is the environment within which an algorithm executes. It specifies which objects exist and the access paths by which they may be reached. An object's extent is that part of the execution history of a program over which the objects exists. Language definitions usually classify

the extent of an object as static (exists during the entire execution history), automatic (exists during the activation of some specified syntactic construct, usually a block or procedure), unrestricted (comes into existence at a programmer-specified point and exists for the remainder of the execution history), or controlled (comes into existence at a programmer-specified point and ceases to exist at another programmer-specified point). Storage allocation policies of the implementation are determined by the extent rules of the language. Direct access to an object is determined by the scope rules and parameter mechanisms of the language. Scope rules are normally based on the static structure of the program, rather than the execution history. They dictate the need for additional explicit linkage information in the storage allocated during execution and also affect the treatment of objects of unrestricted extent. (When an object of unrestricted extent can never be accessed again there is no point in retaining its storage.)

1.2 Target Machine Characteristics

As in the case of the source language, each target machine has a single defining document. This document is the hardware reference manual or data sheet provided by the manufacturer. The compiler writer must understand the target machine in a different sense than he understands the source language: He must know what constitutes an elegant target machine program, whereas he need only be concerned with legal source language programs. (Here, "elegance" is used in the sense of mathematics, to indicate an economic use of resources in the achievement of a goal.) This understanding is developed through experience, perusal of existing code, and (if possible) discussion with the machine designers. Tutorial descriptions of the target machine may help the compiler writer to develop a feel for the architecture and operations, but they should never be used in the design of the compiler itself. The purpose of this section is to outline the important characteristics of current machines, providing a road map for the study of a machine definition. (Reference 3 provides a more complete treatment.)

It is best to begin the study of a target machine by determining its storage classes (the mechanisms it provides for retaining the values of data objects) and access functions (the mechanisms it provides for describing operands). Table 1 summarizes the storage classes found in most current machines.

Every machine has a "main memory" that is used for long-term storage of data objects. The elements of the base, index, and accumulator storage classes are often larger than the elements of the main memory. Thus, when information is moved between the main memory and an element of another storage class, several main memory elements may participate. These elements have contiguous addresses, and the entire sequence is referred to by the smallest of the contiguous addresses. There may be an alignment constraint on the address of the sequence, requiring that it be divisible by some integer greater than 1 (usually a power of 2).

It is convenient for the compiler writer to divide the set of all machines into two categories, based on the characteristics of the base, index, and accumulator storage classes: Register machines have a smaller auxiliary memory whose elements are called registers and whose indices are called register numbers. Registers can be used as operands in main memory access functions, and computation can be carried out on them. Thus, they constitute the base, index, and accumulator storage classes of the machine. The

TABLE 1 Typical Storage Classes

Memory	An array of equal-sized storage elements, indexed by the integers $[0, N - 1]$. The index of a memory element is called its address. Any element in the array can be accessed via its address.
Stack	An array of storage elements, accessible only in a last-in, first-out manner.
Base	A storage element capable of containing a memory address and capable of being used as part of an access function.
Index	A storage element capable of containing the integer offset of a component of a composite object and capable of being used as part of an access function.
Accumulator	A collection of overlapping storage elements on which computation can be carried out.
Condition	A storage element consisting of a set of binary flags describing status information.

Motorola 68000 family provides an example of a register architecture. Accumulator machines have distinct elements for the base, index, and accumulator storage classes. The Intel 8086 family provides an example of an accumulator architecture.

Table 2 summarizes the access functions provided by typical machines. The value yielded by each of these access functions might be used directly as an operand (immediate interpretation), it might be used as a main memory address to access the operand value (direct interpretation), or it might be used as a main memory address to access the address of the operand value (indirect interpretation). Not all of the access functions of Table 2 may be available on a given machine, and not all of the interpretations may be possible. Moreover, different instructions may permit different access functions and interpretations for their operands.

The elementary objects of most machines are bit patterns of some fixed lengths (e.g., 8-bit bytes, 16-bit words); composite objects are ordered tuples, interpreted as collections of smaller patterns (e.g., character strings). Every machine operation interprets its operands in a specific way, and usually the only constraints on application of a particular operation to a particular operand are operand size and location. Thus, the set of distinguishable operand types on a target machine is almost always smaller than the set of distinguishable operand types for a source language.

Because all machine objects are bit patterns, every elementary type is finite and its values are in one-to-one correspondence with the natural numbers in $[0, 2^n - 1]$ (n is the number of bits in the bit pattern). A particular operation may or may not interpret the bit pattern according to this correspondence. For example, integer arithmetic operations may use a sign-magnitude, ones complement, or twos complement interpretation, whereas floating-point operations will use a more complex interpretation [4].

TABLE 2 Typical Access Functions

Constant	The result is a constant value held in the instruction.
Register	The result is the value held in a storage element of the base, index, or accumulator class.
R+C	The result is the sum of the value held in a storage element of the base or index class and a constant value held in the instruction.
R+R	The result is the sum of the value held in a storage element of the base class and a value held in a storage element of the index class.
R+R+C	The result is the sum of the value held in a storage element of the base class, a value held in a storage element of the index class, and a constant value held in the instruction.

In general, actions of the target machine are composed serially. Collateral composition is also used, especially when coprocessors are present. (A typical example is the interaction between a microprocessor and a floating-point coprocessor.)

2. COMPILATION TASKS

The primary task of a compiler is to produce a target language algorithm that is equivalent to a given source language algorithm. Secondary tasks are to produce a good (as opposed to simply correct) target language algorithm and to verify that the given text was, in fact, a valid source language algorithm. I shall discuss the two secondary tasks separately, not because they can be grafted onto an existing compiler that does not undertake them, but because an appreciation of how a compiler goes about its primary task is necessary to understand approaches to the secondary tasks.

2.1 Basic Decomposition

Table 3 shows the hierarchy of subtasks making up the decomposition we shall discuss in this section. Analysis is the process of understanding the source language algorithm; synthesis is the process of constructing an equivalent target language algorithm. The analyzer's understanding of the source program is embodied in a "structure tree," whose shape is determined by the abstract syntax of the source language [5]. This tree is decorated with attributes that represent contextual information. It is important to note that the structure tree is a conceptual object that may or may not have a physical embodiment in the implementation of the compiler. In this section, I am concerned only with the logic of the compiler, not with any physical realization of that logic. Section 3 will discuss compiler implementation strategies.

TABLE 3 Decomposition of the Compilation Task

- Analysis
  - Structural analysis
    - Lexical analysis
    - Parsing
  - Semantic analysis
- Synthesis
  - Code generation
    - Execution order determination
    - Instruction selection
    - Resource allocation
  - Assembly
    - Instruction encoding
    - Address resolution

Structural analysis establishes the skeleton of the structure tree for the given source program, whereas semantic analysis decorates that skeleton with the appropriate attributes. For example, consider the Pascal program of Figure 1. The basic symbols of the language are recognized by the lexical analyzer. As in most languages, the basic symbols are classified as identifiers, denotations, and delimiters. Identifiers are given arbitrary meanings by the programmer, and the compiler must uniquely identify each identifier. A denotation is a representation of a particular value in the universe provided by the source language, and the compiler must know the value it represents. Delimiters are representations of operators in the source language or indicate structure in the linear sequence of basic symbols. They determine the actions taken by the parser. Figure 2 shows the basic symbols of the program given in Figure 1. It has no denotations, but there are identifiers (indicated by *id*) and delimiters.

```
FUNCTION GCD(i,j: integer): integer;
   (* Compute the greatest common divisor
     On entry-
       i,j≠0
     On exit-
       GCD=Greatest common divisor of i and j
   *)

   BEGIN
   WHILE i<>j DO
     IF j<i THEN i:=i—j
     ELSE j:=j—i;
   GCD:=i;
   END
```

FIGURE 1 A Pascal program.

FUNCTION	BEGIN	id(*i*)	id(GCD)
id(GCD)	WHILE	:=	:=
(	id(*i*)	id(*i*)	id(*i*)
id(*i*)	<>	–	END
,	id(*j*)	id(*j*)	;
id(*j*)	DO	ELSE	
:	IF	id(*j*)	
id(integer)	id(*j*)	:=	
)	<	id(*j*)	
:	id(*i*)	–	
id(integer)	THEN	id(*i*)	

FIGURE 2 The basic symbol sequence of Figure 1.

The parser identifies the relationships among the basic symbols and groups them into phrases. Each phrase corresponds to a node in the structure tree. During the parse, a structure tree node is built each time a phrase is recognized. A portion of the structure tree for the program of Figure 1 is shown in Figure 3. Note that the leaves of Figure 3 that correspond to identifiers are decorated with attributes, giving the particular identifiers those leaves represent. These intrinsic attributes are derived by the lexical analyzer, as indicated in Figure 2, and attached to the leaves as the parser builds the structure tree. Denotation values are also intrinsic attributes.

The semantic analyzer accepts the structure tree, decorated only with intrinsic attributes. It usually begins by decorating the root with an attribute that represents execution context. All of the other attributes are then computed on the basis of these initial values and rules that are associated with particular source language phrases. For example, in Figure 3 the intrinsic attributes of the leaves are the internal representations of *i* and *j*, and the root is decorated with the environment—the set of identifiers defined in GCD and any enclosing procedures. The environment attribute

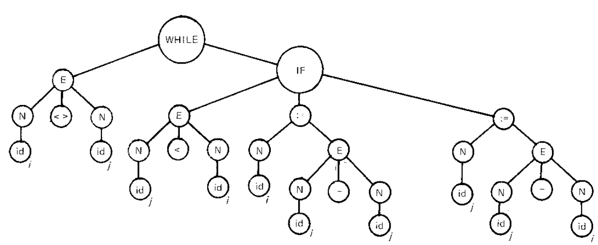


FIGURE 3 A portion of the structure tree for the program of Figure 1.

is propagated unchanged to the leaves, where the identifiers can be looked up in it. It contains information derived from their declarations that defines them as integer variables, local to the current procedure. This information can then be used to derive the types of the objects represented by the name nodes, and that type information, in turn, is used to identify the "—" as an integer subtraction operator (instead of, say, a set difference).

Code generation uses the meaning of the source language algorithm embodied in the structure tree to determine an equivalent target language algorithm. Normally, the first step is to determine an appropriate execution order. Execution order may be fixed for a particular construct (e.g., the condition of an IF must be evaluated before either of the controlled statements), or it may depend upon the relative "complexity" of the children of a node. Complexity information must be computed and attached to the structure tree in the form of attributes, exactly as the information for semantic analysis is derived.

The structure tree is traversed in a manner consistent with the desired execution order, and attribute information available at the nodes is used to select target machine operations and access functions. Figure 4 shows the result of this process on the structure tree of Figure 3, assuming the Intel 8086 as the target computer. (Symbolic assembly code is used to describe the target algorithm in Fig. 4, but this should not be taken to mean that the compiler actually represents the results of code generation in this manner.) The value parameters *i* and *j* occupy space on the stack, and for clarity their names are used to denote the offsets of their locations from the stack frame base register BP; these offsets would be computed by the compiler and placed in the algorithm as numbers.

A target algorithm created during code generation is an abstraction of the actual target program in two ways: Its instructions may be generic forms, and it does not specify final jump target addresses. The second of these abstractions is a consequence of the first, which is illustrated in Figure 4. The Intel 8086 has seven kinds of move instruction, each with a distinct binary encoding and a distinct layout for operand information. These distinctions are uninteresting for code generation because the effect

```
      JMP    L1
L2:   MOV    AX,[BP].j
      CMP    AX,[BP].i
      JGE    L3
      MOV    AX,[BP].i
      SUB    AX,[BP].j
      MOV    AX,[BP].i
      JMP    L4
L3:   MOV    AX,[BP].j
      SUB    AX,[BP].i
      MOV    AX,[BP].j
L4:
L1:   MOV    AX,[BP].i
      CMP    AX,[BP].j
      JNE    L2
```

FIGURE 4 Target algorithm for Figure 3.

of all of them is the same—to move an object from one storage element to another. They are encoded differently only to reduce the average length of a program. Thus, the code generator produces only the generic MOV instruction. Because a generic instruction may be replaced by encodings that have different lengths, use of generic instructions makes it impossible for the code generator to determine the final addresses for instructions and, hence, the values of jump targets. The assembler is therefore responsible for encoding the abstract target algorithm in a form usable by the control unit of the target machine or suitable for further processing by a linker or loader.

2.2 Optimization

It is usually possible to realize the semantics of a given algorithm in many ways. Some of these realizations are cheaper (relative to some given cost function) than others. An optimizing compiler is one that devotes a portion of its resources to a search for a cheap realization. In practice, the search for a true optimum is too costly; the compiler incorporates a fixed sequence of transformations that leads to useful improvement in commonly occurring cases. The primary goal is to compensate for inefficiencies that arise from the characteristics of the source language, not to lessen the effects of poor coding by the programmer.

Because optimization is a search for the best realization of a given algorithm on a given machine, there are no "machine-independent" optimizations. Elimination of common subexpressions is often considered to be machine independent because repeated computations of such expressions are clearly redundant. Most common subexpressions arise in address calculations, however, and the access functions provided by a particular target machine often make recomputation cheaper than saving the result of a previous computation.

The boundary between "optimization" and "competent code generation" is fuzzy: Good local code can be generated by producing simple, uniform instruction sequences and then postprocessing the result, or by tailoring each instruction sequence to the environment in which it will operate. For the purposes of this discussion, I will consider the transformation of the structure tree as code generation. If this transformation results in a physical data structure that is further manipulated before being converted into a target tree, I will consider that manipulation to be optimization. This is an arbitrary definition, but it is a useful one when designing a compiler because it tends to separate concerns.

Optimization techniques can be classified according to the program regions that they consider as units:

- Basic block: A code sequence in which each instruction except the first has a unique predecessor and each instruction except the last has a unique successor.
- Extended basic block: A code sequence in which each instruction except the first has a unique predecessor.
- Strongly connected region: A code sequence in which there is a control path from every instruction to every other instruction.
- Procedure: A code sequence whose execution history constitutes the extent of some set of programmer-defined objects.
- Program: The entire algorithm.

Basic blocks (and extended basic blocks) are maximal sequences having the given property, whereas strongly connected regions and procedures are minimal sequences having the given property. (Once a strongly connected region has been optimized, it is considered atomic, and the focus shifts to any enclosing strongly connected region.) Local optimization considers basic blocks or extended basic blocks, whereas global optimization considers strongly connected regions, procedures, or the entire algorithm.

A typical optimization strategy for basic blocks is to first determine the computations that yield distinct values. This information is used to decide which computations must actually be carried out in the basic block [*6*]. Each computation is given a unique name and is linked to all of its uses. (If there is no information about code that will follow the basic block being optimized, the compiler assumes a pseudo-use for every programmer-defined variable at the end of the basic block.) For example, consider the basic block beginning a L3 in Figure 4 and ending just before L4. The two address computations, [BP].*j*, both yield the same value; therefore the three instructions can be replaced by two, as shown in Figure 5. By applying the same process to the extended basic block beginning at L2 and ending just before L3, the compiler notes that the second move into AX yields the same value as the first and, hence, can be eliminated.

In most programming languages it is possible to refer to the same object by more than one name. A simple example of such aliasing is the use of a global variable as an argument to a Pascal procedure that expects a VAR parameter: Within the procedure the object can be accessed either via its global name or via the name of the corresponding parameter. The compiler must realize that an assignment to the global variable may change the value of the parameter (and vice versa) if it is to determine the computations within the procedure that yield the same values. Analysis of aliasing is dependent upon the source language and constitutes one of the most difficult of the optimizer's tasks [*7, 8*].

Optimization over a strongly connected region involves a flow graph of the region: a graph whose nodes are basic blocks and whose arcs show the connectivity of those basic blocks. Figure 6 gives a flow graph for the strongly connected region of Figure 4. The basic blocks are named by their labels for clarity; the unlabeled block is the one beginning just after JGE. Again, one of the optimizations that can be carried out over a region

```
        JMP     L1
L2:     MOV     AX,[BP].j
        CMP     AX,[BP].i
        JGE     L3
        MOV     AX,[BP].j
        SUB     [BP].i,AX
        JMP     L4
L3:     MOV     AX,[BP].i
        SUB     [BP].j,AX
L4:
L1:     MOV     AX,[BP].i
        CMP     AX,[BP].j
        JNE     L2
```

FIGURE 5 Local optimization of Figure 4.

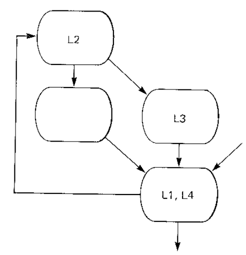


FIGURE 6 Flow graph for Figure 4.

is to determine which computations are redundant. The comparison in block L1 is identical to the comparison in block L2 if the operands are reversed. Moreover, this comparison takes place in every predecessor of L2 (because L1 is the only predecessor of L2). Therefore, it is redundant in L2 and can be eliminated. But this eliminates the entire computation carried out by L2, so the basic block itself disappears. Its only remnants are the two arcs leaving it, which now leave L1.

In addition to detecting redundant computations, optimization over a strongly connected region can detect values that do not change over the region and provide more accurate information about the values that will be used after exit from each basic block. The process begins with summaries of the properties of each basic block. The results of these summaries are then propagated to all basic blocks by an iterative algorithm [*9*]. Redundant computations can be eliminated, and constant computations are moved to the region's entry points [*10*]. Finally, the basic blocks in the region are locally optimized, as discussed above. Figure 7 shows the results for the program of Figure 4.

```
        JMP     L1
L3:     SUB     [BP].j,AX
L1:     MOV     AX,[BP].i
        CMP     AX,[BP].j
        JEQ     L5
        JLT     L3
        SUB     AX,[BP].j
        MOV     [BP].i,AX
        JMP     L1
L5:
```

FIGURE 7 Global optimization of Figure 4.

The purpose of optimizing over a procedure is to assign user-defined variables to registers. The most general approach is to allow all objects, both those defined by the programmer and those defined by the compiler, to compete for registers on an equal basis. Every object's extent is determined, and an interference graph constricted: The interference graph has one node for every object, and two objects are connected by an arc if their extents overlap. Figure 8 shows an interference graph for the program of Figure 7. Because AX and *i* do not interfere, they can be implemented using the same target machine resources. The resulting program fragment is given in Figure 9.

2.3 Error Recovery

Errors can be detected by every one of the subtasks listed in Table 3. Those detected during analysis are violations of the source language definition, whereas those detected during synthesis are violations of constraints imposed by the mapping from source language to target machine. (Examples of the latter are the use of unimplemented language features and requests for more resources than the target machine provides.) Unfortunately, the compiler cannot generally determine the cause of the error but can only note some anomaly that is a symptom of the error. For example, if a Pascal compiler discovers an undeclared identifier, it might be a symptom of a misspelling, an omitted declaration, or a corruption of the program's structure that places the use outside of the scope of the declaration.

There are three levels of response that the compiler might make to detect an anomaly:

1. Report: Provide the user with an indication that an anomaly has been detected. Specify the symptom in understandable terms, precisely locating it with respect to the text of the source program [*11*], and possibly attempt to diagnose the underlying error.
2. Recover: Make the state of all internal data structures consistent and continue in an attempt to detect further anomalies.
3. Repair: If the error diagnosis is very likely correct, make an appropriate alteration of the program or data and continue.

Under no circumstances should the compiler be permitted to crash, because in so doing it is likely to suppress all indication of the error.

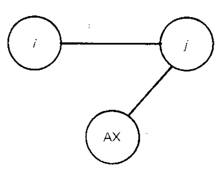


FIGURE 8 An interference graph for Figure 7.


```
        JMP    L1
L3:     SUB    [BP].j,AX
L1:     CMP    AX,[BP].j
        JEQ    L5
        JLT    L3
        SUB    AX,[BP].j
        JMP    L1
L5:
```

FIGURE 9 Assigning *i* to AX in Figure 7.

Every compiler must report all errors that it detects. Recovery is possible in almost all cases, but care must be taken to avoid spurious reports based on an incorrect recovery. Repair should not be attempted unless the user specifically requests it, because it may well result in a program that does not behave as the user expects. Such a program is dangerous because it could destroy or corrupt important data files when run in a production environment.

The influence of error recovery on the design of a compiler is pervasive, and it is a necessary debugging tool during construction of the compiler itself. All error reports should be made via a single routine that accepts four pieces of information:

1. The severity of the error.
2. A definition of the error message text.
3. A source text position for the error report.
4. An indication of the compiler code that detected the error.

The routine saves the information, forming all of the reports into a list sorted by source text position, regardless of the order in which they were actually made. (At the end of the compilation, this list can be merged with the source text or written into a file for processing by a separate tool such as an intelligent editor.) Depending upon the severity of the error, the routine will either return normally or terminate the compilation without returning. This behavior allows the same routine to report both violations of internal compiler assertions and user errors. A single reporting mechanism for all abnormal behavior simplifies the compiler's code and guarantees that every report will actually be output.

3. COMPILATION STRATEGIES

A number of general strategies are available for implementing a compiler. They vary in the implementation effort required, the quality of the generated code, and the cost of the compilation itself. It is important that the compiler designer settle upon a strategy appropriate for the environment in which the compiler is to operate, the compiler construction tools at his disposal, and the budgeted implementation effort. All tactical decisions must be made within this strategic framework. The compiler construction literature unfortunately gives short shift to the distinction between strategy and

tactics, concentrating on the latter at the expense of the former. The result is that there are a large number of algorithms and data structures that simply do not fit together to produce a coherent compiler.

The basic strategic decisions to be made involve the flow of information within the compiler and the physical decomposition that will be used to implement the logical decomposition of Table 3. These decisions are influenced by the tools at hand to aid in the construction process. Two case studies that clearly illustrate the interaction of information flow and physical structure in compilers for small machines are the descriptions of the GIER ALGOL compiler [12] and the Concurrent Pascal compiler [13]. The first of these is also analyzed in Ref. 2, Chapter 14, along with the Zurich Pascal compiler (on which most microcomputer Pascal compilers are based) and the FORTRAN H compiler (which illustrates optimization techniques). In the following subsections, I shall outline four specific strategies and summarize the tactics that support them, giving references to appropriate literature. These strategies are by no means the only ones possible. They are all economically viable, however, and provide useful starting points for detailed design.

3.1 Classical One-Pass Compiler

If the source language is suitably defined, a program can be checked for adherence to the language definition in a single pass over the text without retaining a representation of the program in memory. (The Pascal standard [14, 15] is an example of such a definition.) Often the entire translation can be accomplished as the source program is being checked, although the quality of the generated code may leave something to be desired. The result is a fast compiler with a relatively simple structure.

Classical one-pass compilers are usually implemented by hand as collections of mutually recursive procedures. The main control algorithm is a set of procedures corresponding to the source language grammar [2, 16]. There is a separate collection of procedures to manage definitions [17]. Instruction selection is done within the control procedures, using a separate module to simulate the state of the target machine and allocate registers [18]. The selected instructions may be written out in symbolic assembly code or in the format required by a linker or loader.

A lexical analyzer generator [19, 20] and a parser generator [21, 22] can be used to implement the structural analysis subtask of a classical one-pass compiler. The analyzer generated by these tools will not retain a representation of any part of the program during analysis. Other commonly available tools build compiler components that do retain representations of parts of the program and are thus inappropriate for the classical one-pass strategy. Another strategy, taking advantage of the retention to improve code quality, should be used with such tools. Similarly, if a language definition requires retention of some parts of the program in order to complete semantic analysis, a strategy that makes use of this retention should be employed.

The Zurich Pascal compiler is a good example of the classical one-pass strategy. A case study of this compiler can be found in Ref. 2, Section 14.2.2. There is an excellent account of its development in German [23], and two partial descriptions in English [24, 25].

3.2 Tree Rearrangement

If the target machine is a register machine, intermediate results within an expression can be kept in registers. It can be shown [26] that the order in which the subexpressions of an expression are evaluated is crucial to the number of registers that will be needed to hold intermediate results. By computing an integer attribute that stimulates the number of registers required to compute the subexpression rooted at each node of the structure tree, the compiler can determine the best evaluation order. It can also determine exactly those expressions that must be stored in memory due to an insufficient number of target machine registers [27, 28]. A compiler design based on these results must provide for retention of expression subtrees until the order in which to generate code for them has been determined.

The register estimation attribute is effectively a register allocation, but it does not assign particular registers to intermediate results. Register assignment is still carried out by a target machine simulator and register allocator, as in the case of the classical one-pass compiler. The register allocator is simplified somewhat from that case, because it never needs to "spill" (unexpectedly free) a register.

Reference 27 describes an algorithm for actually assigning the registers as part of the attribution process, but it is better not to do so when compiling certain common source languages. A language like Pascal, for example, has static scoping rules that require rather complex run-time storage accesses [29, 30]. The storage access mechanism is not explicit in the source program and, therefore, is invisible during the attribution process. By allowing the machine simulator to handle allocation of the registers necessary for such storage accesses [2], the code quality can be improved.

Tools that produce parser-driven code selectors [31, 32] can be used to implement the instruction selection subtask of a tree-rearranging compiler. The code selector generated by these tools assumes a prefix linearization of an assignment statement or a condition; the entire component must be retained in order to produce that form. Effective tree rearrangement also requires retention of a complete assignment statement or condition, so these tools are well matched to the tree rearrangement strategy.

3.3 Extended Basic Block Optimization

The embodiment of the source program is a structure tree; every intermediate result in a tree is used exactly once. Certain source language constructs, such as array references, generate repetitious subtrees. For example, the following FORTRAN statement, used in solving three-dimensional boundary value problems, generates the subexpression $I+d_1\times(J+d_2\times K)$ seven times:

```
A(I,J,K) = (A(I,J,K-1) + A(I,J,K+1) +
            A(I,J-1,K) + A(I,J+1,K) +
            A(I-1,J,K) + A(I+1,J,K)) / 6.0
```

(Here, d_1 and d_2 are the first two dimensions of A). Saving this value in a register instead of recomputing it would improve the generated code.

It is relatively easy to keep track of such common subexpressions within a single sequence of operations. Instead of selecting target machine instructions, as in the previous cases, the code generator produces tuples. A tuple is a specification of a target machine operator and appropriate operands (which may be constants or the names of previously generated tuples). These tuples are passed to a module that retains them until a label is reached. The module also recognizes tuples that have the same value, eliminating the second and subsequent occurrences, by a process known as value numbering [2].

When a label is reached, the tuple buffer contains an extended basic block. As the tuples are entered into the buffer, the compiler notes where each value is created and how many times it is used. The code can then be rearranged to minimize the extent of each computation's value [33, 34], and peephole optimization [35] applied to the rearranged code. If the target machine has a register architecture, or if there are several elements in the base and index storage classes, these resources are referred to symbolically until the final sequence is obtained.

It is important to note that all address calculations, including those that are not explicit in the source program, must be represented explicitly in the tuple sequence. This means that the code generator must insert addressing tuples instead of leaving that task to the target machine simulator, as was done under the previous strategy. Also, the register allocation has more information available to it because it knows the precise lifetime of each intermediate value. The resource allocation algorithm of Ref. *36* is reasonable in this context.

A peephole optimizer generator [37, 38] can be used to implement the peephole optimization subtask in a compiler that follows the extended basic block optimization strategy. The optimizer generated by this tool performs common subexpression elimination and reordering, as well as peephole optimization. It operates on a representation of the program that is a correct sequence of instructions for an infinite register analog of the target machine. (In other words, the instruction selection subtask has been carried out, but the resource allocation has not.) Compilers using extended basic block optimization often also employ tree rearrangement, although Davidson and Fraser argue that the ultimate code is just as good using the classical one-pass strategy to select instructions for input to the extended basic block optimizer [37].

3.4 Global Optimization

The code generator of a compiler following the global optimization strategy produces tuples from the structure tree, passing them to a computation graph module that retains them. This module also provides facilities for the code generator to mark basic block boundaries and to specify the connectivity of the basic blocks. Typically, the module keeps the tuples in a list and develops a separate, linked data structure for the flow graph [8].

General code motion [39] and strength reduction [40] transformations may be applied to the computation graph, or transformations that are quite language and machine specific [41] may be appropriate. In either case, the result will be a valid computation graph. All decisions regarding redundant computations will have been made, and the computations removed if appropriate. This computation graph can be linearized to minimize the number of unconditional jumps executed inside a loop, and the tuples within each basic block rearranged to minimize the extent of each computation's value. Finally, the operations can be coded, and peephole optimization applied to the resulting instruction sequence.

A key point in the design of a globally optimizing compiler is the specification of the computation graph data structure. Theoretically, it must be able to specify definition and use information for every computation appearing in the source program; practically, the set of such expressions should be restricted to those that offer possibilities for optimization. Such possibilities depend, in part, on the target machine—Ref. *8* gives useful criteria.

The primary reason for considering a procedure or the entire algorithm during optimization is to assign programmer-defined objects to registers. A uniform and intuitively appealing approach is graph coloring [42], alluded to in connection with Figure 8. Unfortunately, this algorithm seems to be quite slow [43]. Another approach is to measure the number of registers needed locally within each basic block and to allocate any extras globally [44]. After the global assignment has been made, the unassigned registers are assigned within each basic block by the techniques of Ref. *36*.

4. THE DESIGN PROCESS

Compiler design begins with specifications of the source and target languages. The earliest and most critical decisions concern the relationship between these languages—the definition of equivalence, mentioned in Section 1. They are embodied in a mapping specification that generally follows the outline of the source language definition [2]. This mapping specification describes one or more target language constructs equivalent to each source language construct. The target representations of a source construct are selected solely on the basis of the characteristics of that source construct. In other words, optimizations stemming from possible relationships among the target language constructs are explicitly omitted from the mapping specification.

A mapping specification defines a correct translation of source code to target code. It contains all of the information needed to design a compiler, carrying out only the basic subtasks listed in Table 3, and provides an insight into the quality of the code that will be produced. The next decisions concern the basic compilation strategy. They are based on the goals of the compiler and the resources available for its development and result in a physical decomposition.

At this point in the design, the information processing requirements of the compiler are known. Standard algorithms are available for all of the components of a compiler that follows any of the strategies presented in Section 3. The general shape of the data structures for all of these strategies is also known, but different mapping specifications will require variations. It is best to begin by specifying the information content of each data structure and devising a particular representation for that information in conjunction with the realization of the processing algorithms.

REFERENCES

1. J. McCarthy, "Towards a Mathematical Theory of Computation," in *Information Processing 1962*, North-Holland, Amsterdam, 1963.
2. W. M. Waite and G. Goos, *Compiler Construction*, Springer Verlag, New York, 1984.
3. J. F. Wakerly, *Microcomputer Architecture and Programming*, John Wiley & Sons, New York, 1981.
4. IEEE, *Binary Floating Point Arithmetic*, New York, 1985.
5. W. M. Waite and L. R. Carter, *An Analysis/Synthesis Interface for Pascal Compilers*, Software-Practice & Experience, 11, 769-787 (August 1981).
6. W. A. Wulf, R. K. Johnsson, C. B. Weinstock, and S. O. Hobbs, *The Design of an Optimizing Compiler*, American Elsevier, New York, 1975.
7. D. S. Coutant, "Retargetable High-Level Alias Analysis," *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages* (January 1986).
8. P. Anklam, D. Cutler, R. Heinen, and M. D. MacLaren, *Engineering a Compiler*, Digital Press, Bedford, MA, 1982.
9. M. S. Hecht, *Flow Analysis of Computer Programs*, Elsevier North-Holland, New York, 1977.
10. E. S. Lowry and C. W. Medlock, "Object Code Optimization," *Commun. ACM, 12*, 13–22 (January 1969).
11. P. J. Brown, "My System Gives Excellent Error Messages—Or Does It?," *Software—Practice & Experience, 12*, 91–94 (January 1982).
12. P. Naur, "The Design of the GIER ALGOL Compiler," *Annu. Rev. Automatic Programming, 4*, 49–85 (1964).
13. A. C. Hartmann, *A Concurrent Pascal Compiler for Minicomputers*, Springer Verlag, Heidelberg, 1977.
14. IEEE, "Pascal," X3.97-1983, New York, 1983.
15. K. Jensen, N. Wirth, A. B. Mickel, and J. F. Miner, *Pascal User Manual and Report*, third ed., Springer Verlag, New York, 1985.
16. R. M. McClure, "An Appraisal of Compiler Technology," in *Spring Joint Computer Conference*, Vol. 40, AFIPS Press, Montvale, NJ, 1972.
17. W. M. McKeeman, "Symbol Table Access," in *Compiler Construction—An Advanced Course* (F. L. Bauer and J. Eickel, eds.), Springer-Verlag, Heidelberg, 1976.
18. W. M. Waite, "Code Generation," in *Compiler Construction—An Advanced Course*, Vol. 21 (F. L. Bauer and J. Eickel, eds.), Springer Verlag, Berlin, 1976.
19. M. E. Lesk, "LEX—A Lexical Analyzer Generator," *Computing Science Technical Report 39*, Bell Telephone Laboratories, Murray Hill, NJ, 1975.
20. V. P. Heuring, "Automatic Generation of Fast Lexical Analyzers," SEG-85-1, Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO, September 1985.
21. S. C. Johnson, "Yacc—Yet Another Compiler-Compiler," *Computer Science Technical Report 32*, Bell Telephone Laboratories, Murray Hill, NJ, July 1975.
22. P. Dencker, K. Dürre, and J. Heuft, "Optimization of Parser Tables for Portable Compilers," *ACM Trans. Prog. Lang. Syst., 6*, 546–572 (October 1084).

23. U. Ammann, "Die Entwicklung eines PASCAL—Compilers nach der Methode des Strukturierten Programmierens," Ph. D. Thesis, Eidgenössischen Technischen Hochschule Zürich, Zürich, 1975.
24. U. Ammann, "The Method of Structured Programming Applied to the Development of a Compiler," in *Proc. Int. Comput. Symp. 1973*, North-Holland, Amsterdam, 1974.
25. U. Ammann, "On Code Generation in a PASCAL Compiler," *Software—Practice & Experience, 7*, 391–423 (1977).
26. R. Sethi and J. D. Ullman, "The Generation of Optimal Code for Arithmetic Expressions," *J. ACM, 17*, 715–728 (October 1970).
27. J. L. Bruno and T. Lassagne, "The Generation of Optimal Code for Stack Machines," *J. ACM, 22*, 382–396 (July 1975).
28. A. V. Aho and S. C. Johnson, "Optimal Code Generation for Expression Trees," *J. ACM, 23*, 488–501 (July 1976).
29. B. Randell and L. J. Russell, *ALGOL 60 Implementation*, Academic Press, London, 1964.
30. M. Griffiths, "Run-Time Storage Management," in *Compiler Construction—An Advanced Course* (F. L. Bauer and J. Eickel, eds.), Springer Verlag, Heidelberg, 1976.
31. R. S. Glanville and S. L. Graham, "A New Method for Compiler Code Generation," in *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, Association for Computing Machinery, New York, January 1978.
32. R. Landwehr, H. Jansohn, and G. Goos, "Experience With an Automatic Code Generator Generator," *SIGPLAN Notices, 17* (June 1982).
33. W. M. Waite, "Optimization," in *Compiler Construction—An Advanced Course*, Vol. 21 (F. L. Bauer and J. Eickel eds.), Springer Verlag, Berlin, 1976.
34. L. R. Carter, "Further Analysis of Code Generation for a Single Register Machine," *Acta Inf., 18*, 135–147 (November 1982).
35. W. M. McKeeman, "Peephole Optimization," *Commun. ACM, 8*, 443–444 (July 1965).
36. F. Luccio, "A Comment on Index Register Allocation," *Commun. ACM, 10*, 572–574 (September 1967).
37. J. W. Davidson and C. W. Fraser, "Code Selection through Object Code Optimization," *ACM Trans. Prog. Lang. Sust., 6*, 505–526 (October 1984).
38. J. W. Davidson and C. W. Fraser, "Automatic Generation of Peephole Optimizations," *SIGPLAN Notices, 19*, 111–116 (June 1984).
39. E. Morel and C. Renvoise, "Global Optimization by Suppression of Partial Redundancies," *Commun. ACM, 22*, 96–103 (November 1979).
40. F. E. Allen, J. Cocke, and K. Kennedy, "Reduction of Operator Strength," in *Program Flow Analysis: Theory and Applications*, (S. S. Muchnik and N. D. Jones, eds), Prentice Hall, Englewood Cliffs, NJ, 1981.
41. R. G. Scarborough and H. G. Kolsky, "Improved Optimization of FORTRAN Object Programs," *IBM J. Res. Dev., 24*, 660–676 (November 1980).
42. G. J. Chaitin, "Register Allocation & Spilling via Coloring," *SIGPLAN Notices, 17*, 98–105 (June 1982).

43. J. Cocke and P. W. Markstein, "Measurement of Code Improvement Alogrithms," in *Information Processing 80*, S. H. Lavington, ed.), North-Holland, Amsterdam, 1980.
44. J. C. Beatty, "Register Assignment Algorithm for Generation of Highly Optimized Object Code," *IBM J. Res. Dev.*, *18*, 20-39 (January 1974).

WILLIAM M. WAITE